



Escuela
Politécnica
Superior

Desarrollo de un videojuego para NES en ensamblador 6502



Grado en Ingeniería Multimedia

Trabajo Fin de Grado

Autor:

Pablo Máñez Fernández

Tutor/es:

Francisco José Gallego Durán

Mayo 2020



Universitat d'Alacant
Universidad de Alicante

Desarrollo de un videojuego para NES en ensamblador 6502

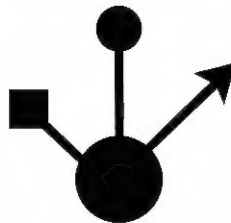
Autor

Pablo Máñez Fernández

Tutor/es

Francisco José Gallego Durán

Ciencia de la computación e inteligencia artificial



Grado en Ingeniería Multimedia



Escuela
Politécnica
Superior



Universitat d'Alacant
Universidad de Alicante

ALICANTE, Mayo 2020

Preámbulo

Por suerte o por desgracias, desde pequeño he crecido junto al mundo de los ordenadores y la tecnología, haciéndome ver, desde los seis años que vi por primera vez la primera GameBoy con el Super Mario Land a sus espaldas, que de una forma u otra tenía que formarme para, en un futuro, intentar trabajar en este sector.

He pasado por grandes baches en mi época universitaria, aunque nada me ha parado y desde que entré ya sabía que quería realizar un videojuego en un futuro Trabajo Final de Grado (TFG), pero ¿en qué plataforma?

El pistoletazo de salida lo dió, hace un año, el concurso CPCRetroDev: Me hizo ver que ahora mismo, por la evolución natural de la tecnología, hacer un videojuego es un tema completamente distinto a como era hace 30 años. De una forma u otra el lenguaje ensamblador me cautivó. Además, siempre he dicho que me habría gustado vivir la época de auge de los ordenadores modernos y las primeras videoconsolas, dado que siempre me ha apasionado este mundo.

Por ello me he retraído unas décadas atrás y me he embarcado en esta aventura de realizar un videojuego para la Nintendo Entertainment System (NES) del que quiero acordarme toda mi vida debido a su valor académico y sentimental.

Agradecimientos

Primeramente, este trabajo no habría sido posible si no hubiera crecido con el acercamiento a la tecnología que he tenido desde pequeño. Va dedicado a esos buenos momentos que paso y he pasado jugando delante de una pantalla con mis mejores amigos. En especial me gustaría agradecer la compañía de todos estos años a Juan, quien ha sido mi compañero de fatigas ingenieril desde el momento que ambos queríamos seguir un camino parecido.

También me gustaría agradecer el trabajo a mi tutor Fran y a todos los profesores que saben como enganchar a los alumnos en sus clases siendo, sin duda, mi punto de partida a la hora de realizar un trabajo de esta índole y a todas aquellas personas que he conocido durante mi etapa en la Universidad y que me han hecho pasar tan buenos ratos, sobretudo a mis compañeros de Gamma Games.

Finalmente, y no menos importante, agradecer todo el apoyo que me ha dado mi familia: Si el día a día de una persona que está en una presión constante y está realizando un trabajo tan importante para él es difícil, no me quiero ni imaginar aquellas personas que lo deben aguantar.

No me olvido de ti Michelle: Muchas gracias por estar siempre a mi lado y aguantarme decir frases y frases sobre conceptos que no entiendes, pero que estás siempre ahí para apoyarme y escucharme.

Sin duda este trabajo quiero dedicarlo a todas aquellas personas que han estado cerca mía durante mi etapa estudiantil desde bachiller hasta la Universidad y que, a pesar de todo, han estado a mi lado apoyándome de una forma u otra.

*A mi familia,
estéis cerca o lejos, sois un gran apoyo para mí*

*A los Gansos,
porque solo tienen que decirme como atacar y defender*

*A Michelle,
gracias por sacarme una sonrisa en todos y cada uno de los días*

*Cualquier memo puede escribir código que un ordenador pueda entender.
Los buenos programadores escriben código que los humanos pueden entender.*

Martin Fowler

Índice general

1. Introducción	1
2. Estado del arte	3
2.1. Super Mario Bros.	3
2.2. MegaMan	5
2.3. NesDev	7
2.4. The Mojon Twins	8
2.4.1. Cheril Perils Classic	9
2.4.2. Cadáveriön	11
2.4.3. Super Uwo!	12
2.4.4. Sir Abadol Remastered Edition NES	13
3. Marco teórico	15
3.1. CPU	15
3.2. PPU	16
4. Objetivos	17
5. Metodología	19
6. Desarrollo	21
6.1. Pintado de sprites	21
6.2. Estructura del programa base	22
6.2.1. Headers de iNES	22
6.2.2. Rutinas de inicio y reinicio	23
6.2.3. Rutinas de inicio del juego y bucle principal	24
6.2.4. Subrutinas del juego y zona de datos	25
6.2.5. Zona de vectores de interrupción y sprites	26
6.3. Compilando el primer programa	27
6.4. Movimiento del personaje principal	27
6.5. Pintado del fondo	28
6.6. Implementación del scroll	30
6.6.1. Aplicando los colores correctamente	34
6.7. Colisiones	36
6.7.1. Cómo calcular las colisiones	37
6.7.2. Integración de las colisiones con el scroll	40
6.8. Enemigos	41
6.9. Objetos y barra de estados	42
6.10. Bank switching para construir una pantalla inicial	44

6.11. Compresión de los datos	45
6.12. Animaciones	46
6.13. Seguimiento del proyecto	47
6.13.1. Iteración 1 - Introducción al sistema NES	47
6.13.2. Iteración 2 - Movimiento alrededor de la pantalla y el nivel	49
6.13.3. Iteración 3 - Primer prototipo jugable	52
6.13.4. Iteración 4 - Optimizando las rutinas y sistema de estados	54
6.13.5. Iteración 5 - Cambiando el estilo visual	58
6.13.6. Iteración 6 - Mejora del diseño del juego	63
6.13.7. Iteración 7 - Videojuego terminado	68
6.13.8. Iteración 8 - Puliendo las transiciones	74
7. Conclusiones	79
A. Lenguaje ensamblador 6502	81
A.1. Set de instrucciones del lenguaje ensamblador 6502	81
A.2. Registros	82
A.2.1. Acumulador	82
A.2.2. El registro X e Y	83
A.2.3. El registro de estados	83
A.2.4. El contador del programa	83
A.2.5. El puntero a la pila	84
A.3. Modos de direccionamiento a la memoria	84
A.3.1. Inmediato	84
A.3.2. Zero-page	84
A.3.3. Absoluto	84
A.3.4. Implícito	85
A.3.5. De acumulador	85
A.3.6. Indexado	85
A.3.7. Indirecto pre-indexado	86
A.3.8. Indirecto post-indexado	86
A.3.9. Indirecto	87
A.3.10. Relativo	87
B. CPU	89
B.1. RAM	89
B.2. Pila	89
B.3. RAM interna	89
B.4. Registros de entrada y salida	89
B.5. Controles	93
B.6. ROM de expansión	94
B.7. SRAM	94
B.8. ROM	94
B.9. Bancos de memoria	94
B.10. Interrupciones	94
B.10.0.0.1. Vector	95

C. PPU	97
C.1. Pattern table	97
C.1.0.0.1. Carga de sprites	97
C.2. Name table	97
C.2.0.0.1. PAL vs NTSC	98
C.2.0.0.2. Mirroring	98
C.3. Attribute Table	99
C.3.0.0.1. Paletas de colores	99
D. Documento de Diseño del Videojuego	101
D.1. Descripción inicial	101
D.2. Control de cambios	101
D.3. Descripción general	101
D.3.1. Contexto e historia	101
D.3.2. Funcionalidades generales	102
D.3.3. Características de los personajes	102
D.3.4. Escenarios	103
D.3.5. Restricciones	103
D.4. Requisitos específicos	103
D.4.1. Mecánicas de los jugadores	103
D.4.2. Mecánicas de los de los objetos y NPCs	103
D.4.3. Técnicas y algoritmos a desarrollar	103
D.4.4. Requerimientos no funcionales	104
D.4.5. Restricciones	105

Índice de figuras

2.1. Sprite 0 utilizado en Super Mario Bros.	4
2.2. Organización de los sprites para la animación del enemigo Goomba	4
2.3. Reutilización de sprites en Super Mario Bros.	5
2.4. Paletas a juego con los poderes de cada nivel en MegaMan	5
2.5. Almacenamiento de los sprites de todos los enemigos	6
2.6. Carga y descarga de los datos de las pattern table en MegaMan	6
2.7. Organización del sprite de MegaMan	7
2.8. Pantallas de inicio y juego de Concentration ROOM	8
2.9. Captura de algunas de las pattern table	9
2.10. Utilización de la pattern table para la aparición de diálogos	9
2.11. Pattern tables utilizadas en Cheril Perils Classic	10
2.12. Cambio entre zonas de Cheril Perils Classic	10
2.13. Comportamiento de los enemigos en Cheril Perils Classic	11
2.14. Organización estatuas y el reloj en Cadáveriön	12
2.15. Pirámide de niveles en Super Uwol!	13
2.16. Barra de estados con la puntuación en Sir Abadol Remastered Edition NES .	14
3.1. Mapa de organización de la memoria de la Unidad Central de Procesamiento (CPU)	15
3.2. Mapa de organización de la memoria de la Unidad de Procesamiento de Imágenes (PPU)	16
6.1. Sprite cargado en memoria de la CPU	21
6.2. Paletas de colores cargadas en memoria de la PPU	22
6.3. ¡Hola mundo!	28
6.4. Representación del mapa en memoria	29
6.5. Primer pintado del fondo	29
6.6. Primer pintado correcto del fondo	30
6.7. $SCROLL = 0$	31
6.8. $SCROLL = 16$	31
6.9. Formas de almacenar la información del mapa	32
6.10. Organización de la attribute table 0	34
6.11. Forma de almacenar la información de las attribute table	36
6.12. Bloques con colisiones en color	36
6.13. Esquema de almacenado del mapa de colisiones en memoria	37
6.14. Cálculo del byte X para las colisiones	38
6.15. Cálculo de la subzona según la coordenada X	39
6.16. Cálculo del desplazamiento en Y del mapa de colisiones	39
6.17. Almacenamiento final del mapa en memoria	40

6.18. Posibles situaciones de los bloques según el scroll	41
6.19. Columnas seleccionadas para la aparición de enemigos	42
6.20. Mostrando la puntuación en la barra de estado	44
6.21. Pattern tables utilizadas	45
6.22. Organización de cada metasprite	46
6.23. Diagrama con los pasos que sigue un sistema de animaciones	46
6.24. Numeración de los distintos frames de la animación	47
6.25. Imagen del primer prototipo	48
6.26. Comportamiento del valor que se introduce en 0x2005	50
6.27. Esquema de funcionamiento del ring-counter utilizado para la lectura de los controles	51
6.28. Esquema de división de la memoria en distintos bancos	52
6.29. Forma nueva de almacenamiento del mapa de colisiones	53
6.30. Desactivación de la llamada a una rutina	55
6.31. Esquema de funcionamiento del sistema de estados	55
6.32. Estado de la memoria en el momento de la creación de un enemigo	56
6.33. Esquema de adición del offset a los enemigos para un movimiento alrededor del nivel	57
6.34. Orden correcto de ejecución de las rutinas en el bucle principal	59
6.35. Sprites definitivos	59
6.36. Diseño inicial del personaje principal	60
6.37. Cambio del estilo general de los niveles	60
6.38. Diseño de la pantalla inicial	61
6.39. Organización de los datos de cada metasprite	62
6.40. Metasprites de los niveles definidos para su uso	63
6.41. Datos de las colisiones y los atributos de cada uno de los sprites	64
6.42. Construcción del mapa en Tiled	64
6.43. Composición de las metatiles en la pantalla de inicio	65
6.44. Muestra del comportamiento nuevo de los enemigos	66
6.45. Esquema de control de la activación del flag carry en la rutina de colisiones . .	66
6.46. Nivel 1 definitivo	67
6.47. Animaciones de los enemigos	67
6.48. Esquema de ejecución de la rutina de animaciones	68
6.49. Indicador del nivel actual en la parte superior de la pantalla	69
6.50. Texto "PRESS START" en la pantalla de inicio	70
6.51. Paletas utilizadas en el parpadeo de "PRESS START" en la pantalla de inicio	70
6.52. Todos los niveles disponibles para jugar	71
6.53. Continuidad entre niveles	71
6.54. Diagrama que explica las comprobaciones sobre el nuevo sistema de movimien- to de los enemigos	72
6.55. Offset que se añaden a la rutina de comprobación de colisiones	73
6.56. Situación especial a la hora de aplicar los offset a los enemigos	73
6.57. Transición de paso entre niveles	74
6.58. Pantalla en blanco y negro por la muerte del personaje	75
6.59. Cambio de la paleta de colores del personaje en su muerte	75

6.60. Letras del abecedario en la pattern table	76
6.61. Comparación de escritura de palabras con y sin constantes definidas	76
6.62. Pantalla final con texto	77
C.1. Representación de los mirror que se realizan dentro de la PPU	98
C.2. Orden de pintado de los sprites del fondo	99
C.3. Agrupación de los colores en las dos paletas	99
D.1. Pantalla de inicio de Noxious Liquid	102
D.2. Personaje principal	103
D.3. Enemigos	103

Índice de tablas

6.1. Obtención de la dirección de memoria destino para dibujar	33
6.2. Obtención de la dirección de memoria de origen para dibujar	33
6.3. Obtención de la dirección de memoria destino para escribir los nuevos atributos	35
6.4. Obtención de la dirección de memoria origen para obtener los nuevos atributos	35
6.5. Tabla que almacena el bit a atacar según la coordenada X	38
6.6. Direcciones de memoria que se obtienen dependiendo del valor de la variable	51
6.7. Explicación de la muerte del personaje por altura	54
6.8. Ejemplo de funcionamiento de una tabla de saltos	58
6.9. Tablas que muestran el error en el funcionamiento del scroll con el nuevo sistema de movimiento	58
6.10. Error con el antiguo sistema de movimiento de los enemigos	71
6.11. Offset aplicados al llegar a las bandas críticas de la pantalla	73
6.12. Condiciones para esconder los sprites de los enemigos cuando no deben aparecer en la pantalla	74
A.1. Set de instrucciones del MOS Technology 6502	81
A.2. Flag disponibles	83
B.1. Organización de la memoria asignada a los sprites en la CPU	90
B.2. Organización de la memoria asignada a los registros de entrada y salida en la CPU	91
D.2. Tabla de control de cambios del GDD	101
D.3. Mecánicas de los jugadores	103
D.4. Mecánicas de los objetos y NPCs	103
D.5. Técnicas y algoritmos a desarrollar	103
D.5. <i>Técnicas y algoritmos a desarrollar (continuación)</i>	104
D.6. Requerimientos no funcionales	104
D.6. <i>Requerimientos no funcionales (continuación)</i>	105

Índice de Códigos

3.1. Forma de comunicación CPU-PPU	16
6.1. Almacenaje de la información sobre los sprites	21
6.2. Headers del estándar iNES	22
6.3. Rutinas de inicio y reinicio	23
6.4. Bucle principal del juego	24
6.5. Subrutinas y datos	26
6.6. Vectores de interrupción y adición de sprites	26
6.7. Como compilar un juego para NES	27
6.8. Tabla de saltos utilizada para el movimiento	28
6.9. Pintado del fondo no optimizado	28
6.10. Instrucciones que inhabilitan el scroll	30
6.11. Rutina que realiza el bank swapping	44
6.12. Datos del tipo de enemigo número uno	56
6.13. Parámetros disponibles para el exportador	64
6.14. Almacenamiento de los datos de los sprites	68
A.1. Ejemplo de instrucciones sobre el registro A	82
A.2. Ejemplo de instrucciones sobre los registros X e Y	83
A.3. Ejemplo de instrucciones que involucran el uso de la pila	84
A.4. Modo de direccionamiento inmediato	84
A.5. Modo de direccionamiento absoluto	84
A.6. Modo de direccionamiento absoluto zero-paged	85
A.7. Modo de direccionamiento implícito	85
A.8. Modo de direccionamiento de acumulador	85
A.9. Modo de direccionamiento indexado	85
A.10. Modo de direccionamiento indexado zero-paged	85
A.11. Modo de direccionamiento indirecto pre-indexado	86
A.12. Modo de direccionamiento indirecto post-indexado	86
A.13. Modo de direccionamiento indirecto	87
A.14. Modo de direccionamiento relativo	87
B.1. Funcionamiento de los controles	93
C.1. Forma de activar las pattern table	97
C.2. Activación de la transferencia de los datos de sprites	97
C.3. Carga de los sprites a las name tables	98
C.4. Carga de las attribute table	100

1. Introducción

Hoy en día la tecnología ha llegado a ese nivel del que alguien nos puede enseñar una película y en ciertos casos no poder distinguir si es real o es generado mediante un programa de ordenador. Lo mismo pasa con los videojuegos: Gracias al avance de la tecnología y al hecho de que cada vez existe mucha más capacidad de procesamiento en los ordenadores, nos lleva a resultados increíbles, claro está, gracias al uso de ordenadores compuestos por potentes tarjetas gráficas y procesadores cada vez más rápidos.

Además hay empresas que realizan motores gráficos para que cada vez más gente pueda realizar este y todo tipo de videojuegos: Aplicaciones totalmente preparadas para que cualquier usuario pueda llegar a realizar uno de éstos. Naturalmente, hablamos siempre de tecnologías modernas y, de algún modo, accesibles al usuario de a pie.

Puesto lo anterior sobre la mesa, ¿Qué pasaría si hoy en día alguien quisiera realizar un videojuego en un sistema de hace 30 años? Se encontraría con unas especificaciones técnicas muy limitadas, como bajo procesamiento y poca memoria disponible por parte de la CPU. Incluso podemos tirar a la basura el concepto de tarjeta gráfica.

Por ello, en este trabajo se recoge el proceso de creación de un videojuego para la NES, su funcionamiento bajo este sistema y la documentación necesaria que indica las entrañas de cada instrucción del lenguaje ensamblador 6502 y cómo afectan éstas a los distintos cerebros la componen.

Debido a la escasa cantidad de información que existe actualmente sobre el sistema NES, este documento no contiene sección bibliográfica, puesto que mi fuente de información ha sido la wiki de NesDev¹: Esta web recoge muchos documentos e información muy interesante y necesaria que he utilizado para realizar todo mi trabajo.

¹http://wiki.nesdev.com/w/index.php/Nesdev_Wiki

2. Estado del arte

A pesar de que la NES tiene más de treinta años, en pleno 2020 su escena de desarrollo de videojuegos es algo que sigue existiendo, aunque en mucha menor medida de lo que podría existir en otras plataformas mucho más actuales, ya que, claro está, este sistema está un tanto desfasado, por lo que le es imposible competir actualmente.

En este capítulo se van analizar técnicamente algunos videojuegos relacionados con el tema de mi proyecto, que se han creado a lo largo de los años, tanto cuando se sacó la consola al mercado como proyectos actuales.

2.1. Super Mario Bros.

Si se habla de videojuegos míticos, el primer Super Mario Bros. de la NES es uno de ellos, sin duda alguna, por el gran impacto que tuvo a la industria, debido al mal estado de ésta en la época y por casi inventar un género relacionado con el famoso fontanero.

Autor: Nintendo Entertainment Analysis and Development

Año de creación: 1985

Características:

Memoria: Al ser uno de los primeros videojuegos que se crearon para este sistema, utiliza el mapper 0, NROM, el más sencillo de todos proporcionando un total de 32kb de memoria para la Memoria de Solo Lectura (ROM) y 8kb para el almacenamiento de sprites en la PPU. Claro está este mapper no permite cambios de bancos durante la ejecución del videojuego, por lo que la memoria que es establecida desde un principio no es modificada durante su ejecución, a excepción de la Memoria de Acceso Aleatorio (RAM).

Barra de estados: Las primeras cuatro líneas del fondo de la pantalla están ocupadas para que se pueda dibujar la barra de estados del juego. En ella se muestra la puntuación y el nivel actual, el tiempo restante y las monedas recogidas. El sprite utilizado para la creación de dicha barra es el sprite con índice 0xFF dentro de la pattern table dedicada para los sprites, es decir, la parte inferior de la moneda que está cambiando de color todo el rato, tal y como se muestra en la figura 2.1.

Un detalle es el cambio de color de dicho sprite el cambio de uno de los colores de la paleta. Concretamente es el índice 3 de la paleta que se utiliza para el fondo.

Movimiento: El control de Mario es muy fluido: Dispone de salto y movimiento lateral variable, con la posibilidad de aumentar su velocidad con la B. Cabe destacar que tiene una deceleración en el movimiento al no pulsar ningún botón de movimiento, izquierda o derecha, que varía según la velocidad que se haya alcanzado justo antes. Además, como se ve a lo largo de los niveles, éstos utilizan un scroll horizontal.



Figura 2.1: Sprite 0 utilizado en Super Mario Bros.

Organización de los sprites para las animaciones: En el caso de Mario, cuando es pequeño, tiene un total de tres frames distintos para su animación, aunque contando el frame intermedio tiene en total cuatro frames. Dichos frames no están organizados de una forma especial en la memoria, ya que se usa una tabla de animaciones con los valores de éstas predefinidos. Un caso distinto es la animación del enemigo Goomba, que a pesar de no utilizar más de cuatro sprites totales su animación se basa en el intercambio y el mirror de los dos sprites inferiores que forman el personaje, tal y como se muestra en la figura 2.2.



Figura 2.2: Organización de los sprites para la animación del enemigo Goomba

Reutilización de los sprites: Con la utilización del mapper NROM, se tiene la desventaja de no poder utilizar más de 256 sprites para el fondo y los propios sprites, por lo tanto ya entran técnicas ingeniosas para poder reutilizar sprites y que los niveles sean variados. Para ello puede observarse ya en el primer nivel como las nubes y los arbustos comparten los mismos índices aunque tengan distinta paleta de colores como se ve en la figura 2.3.

Enemigos: Éstos aparecen durante el transcurso de los niveles y, sin duda, uno de los aspectos más interesantes al contar con una gran variedad de enemigos que van desde los Goombas, que se interponen en tu camino, pasando por Lakitu, que, montado en una nube, lanza bombas desde el cielo, hasta el jefe final Bowser que cuenta con una ráfaga de mortales martillos.

Sobrepaso del límite de 8 sprites por scanline: Aunque durante todo el juego está controlado que no haya más de 4 entidades compuestas por sprites en una misma pantalla, donde se incluye la aparición de enemigos si ya hay suficientes en el momento de aparición, también se utiliza un método para rotar los sprites y cambiar el orden en el que están dentro de la RAM.

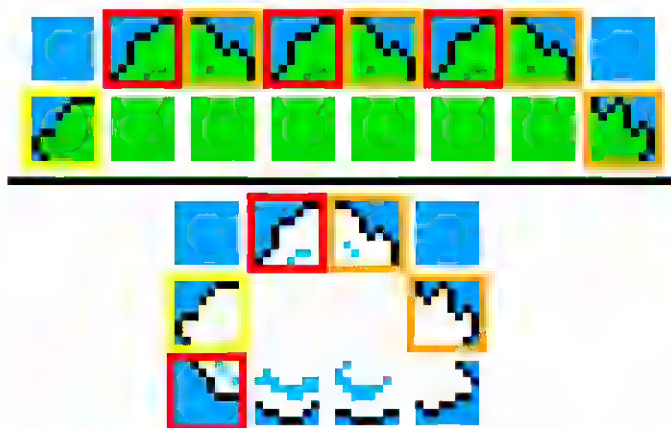


Figura 2.3: Reutilización de sprites en Super Mario Bros.

2.2. MegaMan

RockMan en tierras niponas, es un juego de plataformas donde controlas el personaje con el mismo nombre. Éste tiene la habilidad de moverse de izquierda a derecha, saltar y disparar con su brazo. El juego consta de 7 niveles con su correspondiente jefe final, donde cada uno representa un poder distinto.

Autor: Capcom

Año de creación: 1987

Características:

Memoria: MegaMan utiliza el mapper número dos, el cual permite cambiar entre bancos de 16 kb de memoria de entre un total de 8 o 16, según se desee. Gracias a esto, cada nivel, pantalla de inicio y selección de nivel tienen sus propias pattern tables. Estéticamente, tiene una paleta de colores asociada al poder de cada nivel, como una de colores más fríos para el nivel de Iceman, como puede verse en la figura 2.4.



Figura 2.4: Paletas a juego con los poderes de cada nivel en MegaMan

En cuanto al uso de este mapper, éste tiene una serie de características especiales: La más característica es que no permite el almacenamiento directo en CHR RAM, es decir, dentro de la PPU, como puede permitir el mapper tres por ejemplo. Desde un principio toda la información está almacenada en la ROM y ésta debe ser transferida a la PPU. Una gran ventaja es una mejor organización de los datos

utilizados para los sprites de los personajes y el fondo y la posible compresión de los datos, al estar almacenados desde un principio en la CPU.

Concretamente los datos de todos los sprites están guardados mediante bloques de información fijos de 128 bytes, como se puede ver en la figura 2.5, y son utilizados según se desee.



Figura 2.5: Almacenamiento de los sprites de todos los enemigos

Desarrollo de los niveles: Cada nivel puede ser dividido en ciertas pantallas y, en sí, solo dispone de scroll en el eje horizontal, aunque para ambos lados. En ciertas zonas de cada pantalla hay una pequeña animación que se utiliza como transición para cargar la siguiente pantalla del nivel. Esta transición se realiza tanto utilizando un desplazamiento en vertical como en horizontal.

Estas transiciones son utilizadas para modificar la información que almacenan las pattern table y, por ejemplo, cambiar los sprites que se están utilizando en pequeños trozos de código, tal y como se ha mostrado anteriormente y como se emula en la figura 2.6.

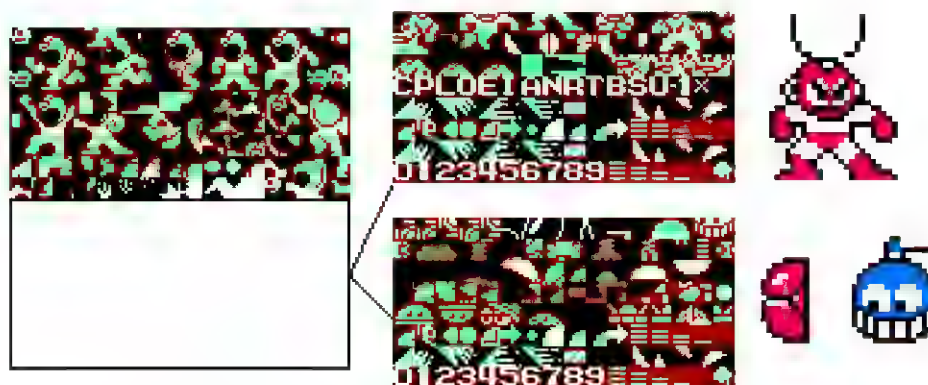


Figura 2.6: Carga y descarga de los datos de las pattern table en MegaMan

Personaje principal: MegaMan está constituido por un mínimo de 13 sprites: 12 para el aspecto del cuerpo, cuando está estático, y uno extra para la cara. El sprite de la cara no va junto al del cuerpo debido a que ésta tiene un comportamiento

distinto al que podría tener el cuerpo en sí, ya que tiene animación propia y utiliza distintos rostros, por ejemplo cuando le golpea un enemigo como se ve en la figura 2.7



Figura 2.7: Organización del sprite de MegaMan

Comportamiento de los enemigos: Los enemigos aparecen en pantalla en el momento que se llega a cierta zona de la pantalla del nivel, por lo tanto, si derrotas a uno de los enemigos y vuelves hacia atrás, el enemigo vuelve a reaparecer en el mismo sitio donde había aparecido en un principio. Si un enemigo sale de la pantalla desaparece automáticamente.

2.3. NesDev

NesDev.com¹ es un portal que recoge una comunidad de usuarios, con un foro activo disponible y multitud de recursos que son muy útiles para empezar a desarrollar para la NES. Concretamente, estos recursos se encuentran en una wiki que se ha ido completando a lo largo de los años, donde se puede encontrar información muy interesante sobre el comportamiento de este sistema, además de todo lo relacionado para el desarrollo de nuevos juegos.

Dentro de esta wiki hay una sección especial donde se recogen proyectos de distintos desarrolladores que se han ido realizando durante los años.

Concentration ROOM: Un juego de cartas que en el que tendremos que girar las cartas que están boca abajo para crear pares y limpiar la mesa: Se controla una especie de puntero, como el del ordenador, con la cruceta de direcciones y con el botón A se da la vuelta a una carta.

Autores: Damian Yerrick, Sara Crickard²

Año de creación: 2010

Características: Este juego no implementa ningún tipo de scroll, siendo siempre una pantalla fija. Solo utiliza una pattern table completa y no son producidos cambios algunos en ésta. En cuanto a la jugabilidad, existen diversos modos de juego entre los que se encuentra: Modo historia, modo solitario, modo dos jugadores y un modo

¹<https://nesdev.com/>

²<http://www.pineight.com>



Figura 2.8: Pantallas de inicio y juego de Concentration ROOM

contra la propia NES, con diversas dificultades entre las que elegir, que altera el número de cartas que hay sobre la mesa.

Super Bat Puncher: Un juego de tipo plataformas en el que se controla un personaje que extiende un puño para combatir a murciélagos.

Autor: Morphcat Games³

Año de creación: 2011

Características: Implementa un tipo de scroll multi-direccional. También se pueden ver, al inicio del juego, una serie de animaciones que, si vemos en las entrañas del juego, están realizadas mediante el intercambio de distintos sprites usados en las pattern table. Entre las utilizadas, se pueden observar una distinta para la animación del inicio del todo, introduciendo el estudio que ha realizado el juego, otra para la propia introducción del juego, con textos, otra más para la pantalla de inicio del juego y, finalmente, una última para el propio juego.

Cabe añadir que, aunque en una captura no es apreciable, en la última, la paleta que pertenece al fondo está cambiando algunos colores para hacer un efecto de animación, en concreto el índice 0 y el 2. Por supuesto, sólo los sprites que estén utilizando esos colores son de los que se hará notar este cambio.

Una característica muy interesante, es la implementación de diálogos en el juego mediante la modificación, en la ejecución del juego, de una parte de la pattern table como el propio diálogo: La pattern table siempre se encuentra en el primer estado y en el momento que salta un dialogo, es cambiada mediante una rutina e impresa, dicha zona, en pantalla, como se ve en la figura 2.10

2.4. The Mojon Twins

Entrando un poco de la escena española dentro del desarrollo de videojuegos en este sistema nos podemos encontrar con The Mojon Twins⁴: Un equipo que actualmente se encuentra en desarrollo con juegos para varios sistemas como Mega Drivem, Spectrum y, como no,

³<http://morphcat.de/>

⁴<http://www.mojontwins.com/>

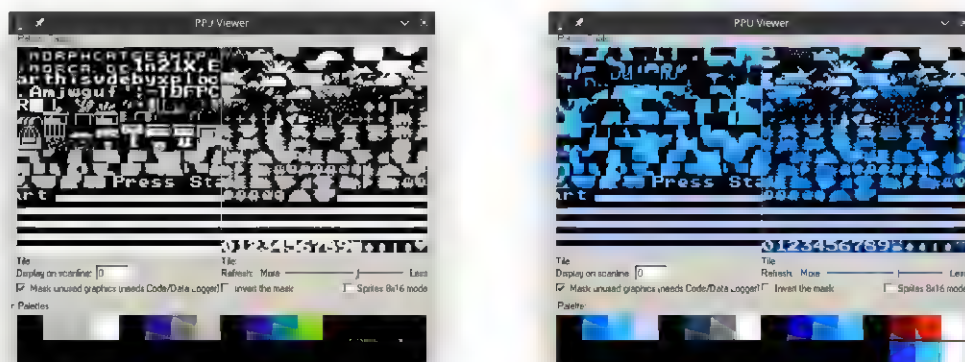


Figura 2.9: Captura de algunas de las pattern table

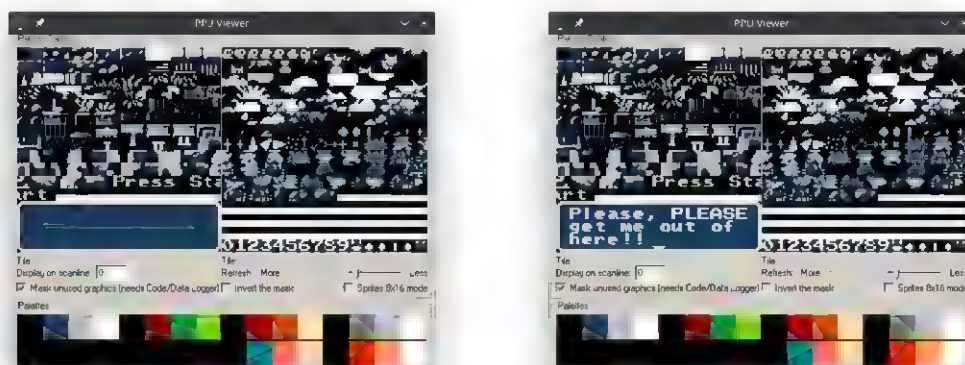


Figura 2.10: Utilización de la pattern table para la aparición de diálogos

NES. Además, para facilitar la creación de estos videojuegos, han desarrollado un pequeño framework/motor en C, **MT Engine MK1 NES**⁵, para crear juegos de forma sencilla y, en parte, como iniciación a la programación de videojuegos para NES.

En esta sección se van a repasar algunos juegos que ha creado este equipo de personas, algunos con el motor mencionado anteriormente.

2.4.1. Cheril Perils Classic

Creado originalmente en 2011 para ZX Spectrum, reprogramado en 2018 con el motor MTE MK1 NES y publicado en 2020, Cheril Perils Classic⁶ es un juego de plataformas en el que hay que recoger unas llaves y pasar por los niveles. Concretamente esta versión es solo una

⁵https://github.com/mojontwins/MK1_NES

⁶http://www.mojontwins.com/juegos_mojonos/cheril-perils-classic-nes-y-sega-8bit/

demo inacabada, en la que se muestran las mecánicas principales del juego original, Cheril Perils, pero mejorada respecto al juego original.

Autor: Mojon Twins

Año de creación: 2011 (Rehecho en 2018)

Características:

Memoria: Este programa no implementa ningún tipo de scroll, todas las zonas en las que se juega son estáticas y se avanza según se explica en el siguiente ítem. En cuanto al mapper utilizado, puesto que no utiliza nada especial, por así decirlo, como scroll o cambio de bancos para intercambiar zonas de memoria, ya que solo se utiliza una pattern table, a pesar de haber distintos niveles, como se ve en la figura 2.11, el escogido será el mapper número cero, con toda la memoria disponible en la ROM y un banco para las pattern table en la PPU.

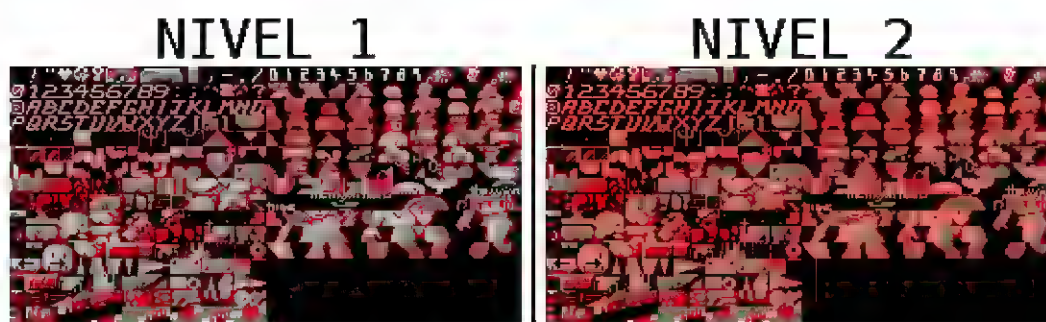


Figura 2.11: Pattern tables utilizadas en Cheril Perils Classic

Pantallas: En esta demo se muestran algunos niveles. Éstos están compuestos por distintas zonas a las que se accede mediante la llegada a ciertos bordes de la pantalla, entonces se dibuja la zona que corresponda tal y como se muestra en la figura 2.12. Para avanzar al siguiente nivel hay que eliminar a todos los enemigos de un nivel.



Figura 2.12: Cambio entre zonas de Cheril Perils Classic

Enemigos: En cada zona tienen un número y tipo de enemigos determinado: Si éstos son eliminados al saltar encima de ellos son eliminados para siempre, en el caso que se vuelva a entrar a la misma zona de nuevo. La forma de eliminarlos es distinta según la dificultad que se elija para jugar al videojuego: Si se elige la dificultad "resonators" habrá que saltar sobre unas plataformas que activan un temporizador para poder eliminar a los enemigos en un corto lapso de tiempo de 10 segundos. Al contrario, en la otra dificultad, más fácil, no existen dichos elementos ya nombrados.

En sí los enemigos no tienen mucha mecánica: Tienen una serie de rutas predeterminadas que siguen una y otra vez como se muestra en la figura 2.13.



Figura 2.13: Comportamiento de los enemigos en Cheril Perils Classic

2.4.2. Cadáveriön

Creado originalmente para ZX Spectrum, este es un port que realizaron The Mojon Twins para la NES utilizando el motor que tienen creado. Cadáveriön⁷ es un videojuego donde Cheril, el personaje principal, está atrapada en una cueva y para escapar debe realizar ciertos puzzles en un límite de tiempo determinado.

Autor: Mojon Twins

Año de creación: 2018

Características:

Objetos: En cada pantalla del mapa hay que llevar un número de estatuas hasta ciertos pedestales. Una vez están todas las estatuas en pedestales, se abre una puerta para

⁷http://www.mojontwins.com/juegos_mojonos/cadaverion-nes/

avanzar al siguiente nivel. Técnicamente, aunque las estatuas puedan moverse, están dibujadas junto al fondo y no son utilizados sprites para éstas. Por lo tanto, en el momento que se empuja cualquier estatua, se deben modificar los bytes que pertenecen a la zona concreta donde estaba y estará.

Por otra parte hay otros objetos que si que se muestran utilizando sprites, como el reloj que te aporta más tiempo para completar el nivel, como se muestra en la figura 2.14.



Figura 2.14: Organización estatuas y el reloj en Cadáverión

Enemigos: En las pantallas de cada nivel hay un cierto número fijo de enemigos. Éstos solo pueden ser esquivados y cualquier toque con ellos hace que se reinicie la pantalla en la que estas jugando. El comportamiento de éstos es un simple movimiento lineal de un sitio a otro. En cuanto chocan con la pared, cambian hacia la dirección opuesta.

2.4.3. Super Uwo!

Super Uwo!⁸ es un videojuego de plataformas en el que se deben recoger monedas a través de los niveles esquivando a los enemigos que hay en ellos y con un límite de tiempo. Cuando este límite de tiempo se acaba, un fantasma va hacia a ti para ponerte las cosas más difíciles. Uwo!, el personaje principal, tiene un total de "dos golpes", por así decirlo, en cada nivel: Cuando es golpeado una vez, su pantalón sale despedido y si le golpean sin su pantalón, pierde una vida de las totales.

Autor: Mojon Twins

Año de creación: 2015

Características:

Memoria: Super Uwo! es un videojuego que salió con el propósito de producirlo físicamente, por lo que el mapper utilizado es el más básico y el más barato de todos: El mapper 0. Por ello no dispone de scroll en los niveles, ya que son estáticos, ni la posibilidad de cambiar de distintos bancos y tener una mayor memoria disponible.

Disposición de los niveles: Los niveles están dispuestos de tal forma que el "escenario principal" es una pirámide donde cada nivel está debajo a la derecha o izquierda según se ve en la figura 2.15.

⁸http://www.mojontwins.com/juegos_mojonos/super-uwo-nes/

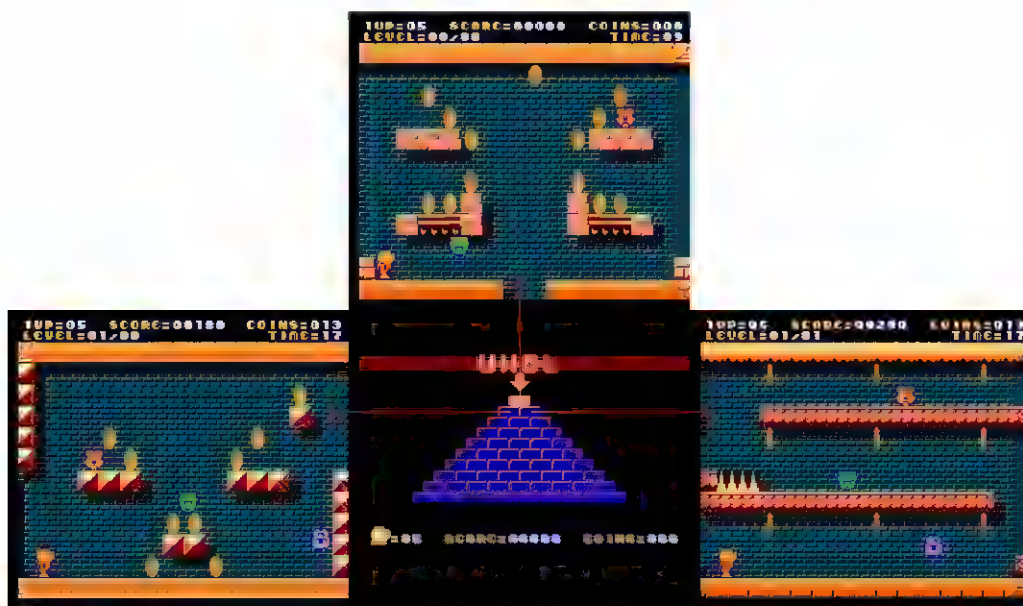


Figura 2.15: Pirámide de niveles en Super Uvol!

Compresión: Para guardar 55 niveles en una memoria de 32 kb, sin contar lógica del juego, han utilizado una técnica de compresión de los mapas en base a la agrupación seguida de tiles iguales, es decir la compresión RLE (Run-Length Encoding). Esta técnica es utilizada mediante un código de dos bytes en el que lleva toda la información de qué byte se repite y cuantos hay del mismo tipo del modo: 07 1A, donde 07 es el número de 1A que debería haber de forma continua en memoria. Esto no tiene por que siempre ser así, puesto que el algoritmo de descompresión y compresión puede ser modificado gusto del usuario.

Se utiliza también una agrupación de los sprites de 8x8 en metasprites individuales de 16x16.

2.4.4. Sir Abadol Remastered Edition NES

Originalmente creado para Spectrum en 2010⁹, la versión original fue el primer juego que The Mojon Twins portaron a NES en 2013¹⁰ y ésta es una remasterización de esa versión original¹¹.

Autor: Mojon Twins

Año de creación: 2015

Características:

⁹[http://www.mojontwins.com/2010/07/07/%c2%a1nuevo-juego-mojono-sir-abadol/Sir Abadol Spectrum](http://www.mojontwins.com/2010/07/07/%c2%a1nuevo-juego-mojono-sir-abadol/Sir%20Abadol%20Spectrum)

¹⁰[http://www.mojontwins.com/juegos_mojonos/sir-abadol-nes/Sir Abadol Original NES](http://www.mojontwins.com/juegos_mojonos/sir-abadol-nes/Sir%20Abadol%20Original%20NES)

¹¹[http://www.mojontwins.com/2017/02/24/nuevo-sir-abadol-remastered-edition-nes/Sir Abadol Remastered Edition NES](http://www.mojontwins.com/2017/02/24/nuevo-sir-abadol-remastered-edition-nes/Sir%20Abadol%20Remastered%20Edition%20NES)

Memoria: Al igual que muchos de los juegos de The Mojon Twins, éste no es menos y utiliza el mapper 0. Una de los aspectos que le diferencian del resto es la implementación de un sistema de scroll en las dos direcciones. Aún así, el mapper utilizado sigue siendo el 0.

Desarrollo del nivel: El objetivo de cada nivel es recoger todas las rosas esquivando a los enemigos. La forma de eliminarlos es colisionar con ellos por arriba, de lo contrario te quitarán una vida. El comportamiento de los enemigos es un movimiento lineal parecido al de los juegos analizados anteriormente.

Éste tiene una barra de estado doble en la que aparecen las vidas, las rosas y las llaves recogidas y, distinto a algunos videojuegos con scroll, está dibujada con sprites y no con tiles de fondo y la utilización del flag de colisión del sprite 0 como se muestra marcado en verde en la figura 2.16, junto a la parte de las pattern table que es utilizada para los sprites.



Figura 2.16: Barra de estados con la puntuación en Sir Abadol Remastered Edition NES

Secretos: Una de las cosas que he descubierto navegando por el código fuente de este juego es la posibilidad de jugar con personajes distintos gracias a la pulsación de cierta combinación "secreta" de botones en el menú principal.

3. Marco teórico

Parte de la tercera generación de consolas, la NES fue una de las videoconsolas que Nintendo sacó al mercado, siendo el primer lanzamiento en 1983 en Japón, la tierra natal de la compañía, donde fue conocida como Nintendo Family Computer (Famicom).

La NES utiliza un microprocesador de 8 bits fabricado por la compañía MOS Technology en 1975: El MOS Technology 6502 (MOS 6502), con un total de 56 instrucciones de memoria. Toda la información relacionada con el lenguaje ensamblador 6502 puede encontrarse en el Anexo A.

Cabe decir que el MOS 6502 es un sistema little-endian: Eso quiere decir que el byte más alto (H) siempre será guardado en memoria en la dirección siguiente que el byte bajo (L).

3.1. CPU

La CPU, es el corazón de todo sistema informático. En concreto, la NES tiene un total de 64 kb de memoria: 0xFFFF direcciones a ser utilizadas. En la figura 3.1 se puede ver un mapa de su organización.

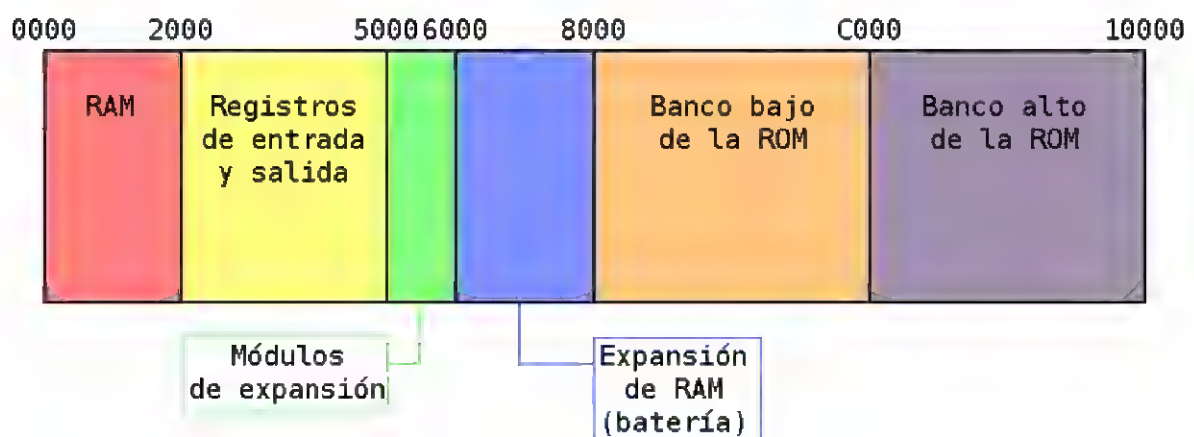


Figura 3.1: Mapa de organización de la memoria de la CPU

Los detalles técnicos de la CPU están explicados en el Anexo B.

3.2. PPU

La Unidad de Procesamiento de Imágenes (PPU) es la unidad de la NES que se encarga del procesamiento de todos los gráficos. Tiene su propio espacio de memoria separado del de la CPU llamada Video RAM (VRAM). Aún así, la PPU tiene un pequeño espacio de memoria que le corresponde dentro de la CPU: El espacio de RAM donde son almacenados los datos de los sprites. En la figura 3.2 se observa un mapa de la organización de su memoria.

Aunque PPU y CPU están separadas, ésta última puede comunicarse con la primera me-

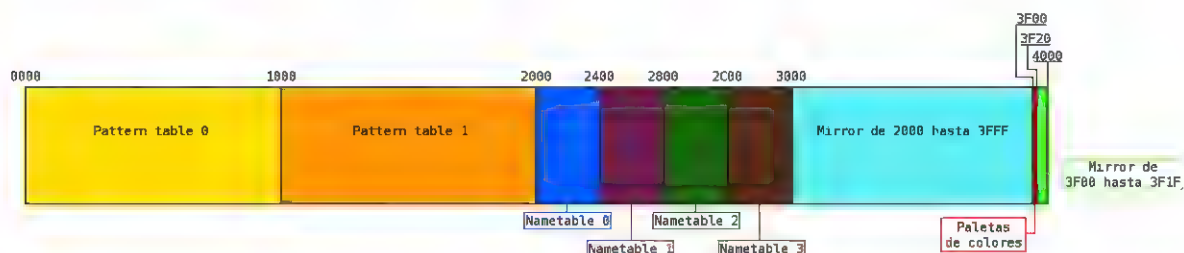


Figura 3.2: Mapa de organización de la memoria de la PPU

dante, por ejemplo, los puertos 0x2006 y 0x2007, explicado en el apartado B.2, para la transferencia de los datos que están almacenados en la CPU y que en realidad pertenecen a la PPU o para establecer ciertas condiciones.

La PPU utiliza direcciones de memoria de 16 bits, como la CPU, y puesto que solo disponemos de registros de 8 bits, se deben realizar dos escrituras en 0x2006 para establecer la dirección de memoria a la que queremos acceder.

Código 3.1: Forma de comunicación CPU-PPU

```
;; Queremos acceder a 0x3F00 para establecer la paleta para el fondo

LDA #$3F ;; Almacenamos en A 0x3F
STA $2006 ;; En la primera escritura se escribe siempre el BYTE ALTO (H)
LDA #$00 ;; Almacenamos en A 0x00
STA $2006 ;; En la segunda escritura se escribe siempre el BYTE BAJO (L)

;; A continuación se realizarían, las operaciones necesarias de
;; lectura para rellenar las paletas correspondientes
```

Los detalles técnicos de funcionamiento de la PPU se encuentran debidamente detallados en el Anexo C.

4. Objetivos

Los objetivos principales de este proyecto son:

- Aprender a utilizar el lenguaje ensamblador para entender como trabajan por debajo los lenguajes de alto nivel actuales y saber como optimizarlos.
- Conocer, entender y dominar la arquitectura de los distintos chips de la NES.
- Diseñar un videojuego desde cero utilizando todos los aspectos posibles que dispone la NES.

Finalmente, debido a la escasez de información que existe sobre este sistema, quiero enfor-car el proyecto al ámbito educativo puesto que puede llegar a ser un punto de salida para futuras personas que deseen programar un videojuego para la NES, mediante la explicación clara y concisa de distintos procesos y características.

Por otra parte, ya como objetivos secundarios, cabe remarcar el objetivo de evolucionar como ingeniero y programador teniendo en cuenta las limitaciones de la consola. También es interesante ver como puedo llegar a desenvolverme con un sistema del que, técnicamente hablando, nunca he tenido contacto con él.

5. Metodología

En este capítulo se va a describir la metodología que se va a utilizar para el desarrollo del proyecto. Concretamente se va a seguir una metodología iterativa donde, dichas iteraciones, serán de aproximadamente un mes de duración con el fin de, al final de éstas, tener un prototipo cerrado que cuente con las características que se plantea implementar.

Los objetivos, tanto primarios como secundarios, de cada iteración vendrá dado por un análisis de lo que se ha conseguido en la anterior. Dicho análisis determinará la forma en la que se va a trabajar, describiendo los pasos que harán falta para la evolución del prototipo de una forma ordenada durante toda la iteración.

Para el correcto desarrollo del proyecto, y de las distintas iteraciones, se van a utilizar las siguientes herramientas:

- **Sublime Text**¹ como editor de textos que se utilizará para escribir todo el código del proyecto.
- **NESASM**² como la herramienta para compilar el videojuego.
- **fceux**³ y **mesen**⁴ como los emuladores para probar el videojuego.
- **Texmaker**⁵ es un programa multiplataforma que me ayudará a realizar esta memoria en L^AT_EX.
- **Toggl**⁶ es un contador de tiempo que monitorea el tiempo de distintas tareas que definas dentro de tu espacio de trabajo.
- **Google Calendar**⁷ y **Google Keep**⁸ como herramientas de definición de objetivos y división del trabajo de cada iteración
- **Github**⁹ para el control de versiones.

¹<https://www.sublimetext.com>

²<https://github.com/camsaul/nesasm>

³<http://www.fceux.com>

⁴<https://www.mesen.ca>

⁵<https://www.xmlmath.net/texmaker>

⁶<https://toggl.com>

⁷<https://calendar.google.com>

⁸<https://keep.google.com>

⁹<https://github.com>

6. Desarrollo

6.1. Pintado de sprites

La NES es un sistema en el que no es posible dibujar un solo píxel en pantalla. La unidad mínima de dibujado, por así decirlo, es el sprite y, más concretamente, un sprite de 8x8 píxeles.

Para poder dibujar un sprite en la NES, primeramente se necesita el conjunto de sprites que vamos a utilizar en nuestro juego. Como aún no se ha definido un tema en concreto, voy a utilizar los sprites del videojuego Super Mario Bros. de la misma plataforma. Más adelante serán intercambiados por los sprites definitivos.

Primeramente tenemos que cargar los sprites en la memoria de sprites de la CPU. Ésta es una parte de la RAM que está compartida con la PPU y guarda el sprite que se va a usar, su posición en la pantalla y el índice relacionado con la paleta de colores con los que se va a pintar el sprite.

000200	80	32	00	80	80	33	00	88	88	34	00	80	88	35	00	88
000210	FE	FE	FE	FE	FE	FE	FE	FE	FE	FE	FE	FE	FE	FE	FE	FE
000220	FE	FE	FE	FE	FE	FE	FE	FE	FE	FE	FE	FE	FE	FE	FE	FE
000230	FE	FE	FE	FE	FE	FE	FE	FE	FE	FE	FE	FE	FE	FE	FE	FE
000240	FE	FE	FE	FE	FE	FE	FE	FE	FE	FE	FE	FE	FE	FE	FE	FE
000250	FE	FE	FE	FE	FE	FE	FE	FE	FE	FE	FE	FE	FE	FE	FE	FE
000260	FE	FE	FE	FE	FE	FE	FE	FE	FE	FE	FE	FE	FE	FE	FE	FE
000270	FE	FE	FE	FE	FE	FE	FE	FE	FE	FE	FE	FE	FE	FE	FE	FE
000280	FE	FE	FE	FE	FE	FE	FE	FE	FE	FE	FE	FE	FE	FE	FE	FE
...
0002F0	FE	FE	FE	FE	FE	FE	FE	FE	FE	FE	FE	FE	FE	FE	FE	FE

Figura 6.1: Sprite cargado en memoria de la CPU

En la figura 6.1 se puede ver como quedaría el mapa de la memoria una vez se han cargado los datos; además se ha marcado en colores los distintos atributos que se pueden asignar a cada sprite, sobre los que se puede encontrar más información en la tabla B.1.

Para ello, se puede realizar mediante accesos directos con el registro A, aunque lo normal sería hacerlo mediante un bucle que cargase los datos. En el fragmento de código 6.1 se muestra la manera en la que pueden estar almacenados los datos en la memoria.

Código 6.1: Almacenaje de la información sobre los sprites

```
DIR_SPRITES:
    .db $80, $32, $00, $80 ;; Sprite 0
```

```
.db $80, $33, $00, $88 ;; Sprite 1
.db $88, $34, $00, $80 ;; Sprite 2
.db $88, $35, $00, $88 ;; Sprite 3
```

Ahora que ya tenemos los sprites cargados en memoria, solo hace falta saber con que colores vamos a pintar éstos. Para ello, debemos comunicárselo a la PPU a través de un registro especial. Pues bien, esta transferencia de memoria se puede realizar a través un bucle parecido al usado con los sprites. Para más información sobre el uso de los registros de entrada y salida en la CPU mirar la tabla B.2.

En la figura 6.2 se observa un pequeño extracto de la memoria que corresponde a las paletas de colores: Concretamente los únicos bytes que sirven son los primeros 32, ya que todos los demás son mirror de las dos primeras líneas.

```
003F00: 22 29 1A 0F 22 36 17 0F 22 30 21 0F 22 27 17 0F
003F10: 00 16 27 18 00 02 38 3C 00 1C 15 14 00 02 38 3C
003F20: 22 29 1A 0F 22 36 17 0F 22 30 21 0F 22 27 17 0F
003F30: 00 16 27 18 00 02 38 3C 00 1C 15 14 00 02 38 3C
003F40: 22 29 1A 0F 22 36 17 0F 22 30 21 0F 22 27 17 0F
003F50: 00 16 27 18 00 02 38 3C 00 1C 15 14 00 02 38 3C
003F60: 22 29 1A 0F 22 36 17 0F 22 30 21 0F 22 27 17 0F
003F70: 00 16 27 18 00 02 38 3C 00 1C 15 14 00 02 38 3C
003F80: 22 29 1A 0F 22 36 17 0F 22 30 21 0F 22 27 17 0F
...
003FE0: 22 29 1A 0F 22 36 17 0F 22 30 21 0F 22 27 17 0F
003FF0: 00 16 27 18 00 02 38 3C 00 1C 15 14 00 02 38 3C
```

Figura 6.2: Paletas de colores cargadas en memoria de la PPU

Después de escribir unas cuantas líneas de código ya tenemos la información que queremos en la memoria, pero... ¿Se puede saber donde va todo eso?!

6.2. Estructura del programa base

Ahora que ya sabemos como guardar los datos de los sprites en memoria, primero tenemos que saber la forma de la que se estructura un programa.

6.2.1. Headers de iNES

A través de una serie de macros, ya establecidas por el estándar iNES, se indican distintas opciones y características que queremos que nuestro juego para NES tenga. Esas opciones se indican como en el fragmento de código 6.2 y es importante sabes que **es lo primero que debe ir en el archivo** y aunque estas macros aceptan mucha más variedad de parámetros, por ahora nos quedamos con estas opciones.

Código 6.2: Headers del estándar iNES

```
;; -INICIO DEL ARCHIVO-
.inesprg 2
.ineschr 1
.inesmap 0
.inesmir 1
;; ...
```

¿Qué es el estándar iNES? Básicamente es el formato que utilizan todos los emuladores actualmente. Mediante las macros que se muestran en el fragmento de código 6.2 se le indican al compilador, en mi caso nesasm, que es lo que va a utilizar nuestro programa.

.inesprg Cantidad de memoria que queremos dedicar a la ROM.

.ineschr Cantidad de memoria que queremos dedicar para los sprites.

.inesmap Mapper que se va a utilizar.

.inesmir Tipo de scroll que se va a implementar.

6.2.2. Rutinas de inicio y reinicio

Las rutinas indicadas en 6.3 son las que, en el momento del encendido de la consola o en el momento que se reinicie, se ejecutarán para una limpieza total de la memoria modificable del juego, es decir, la RAM. Concretamente, el procesador saltará a la etiqueta **RESET** y ejecutará desde cero todo el programa. Además, cabe destacar el uso de las macros **.bank**, que indica el número del banco que empieza, y **.org**, marcando como la dirección para empezar a sobrescribir memoria 0x8000, es decir, el inicio de la ROM.

Desde las rutinas **vblankwait1** y **vblankwait2** se realiza una especie de interrupción NMI casera, ya que en ese momento están desactivadas.

Código 6.3: Rutinas de inicio y reinicio

```
;; ...
;; :::::::::::::::::::::::::::::::::::::::::::::::::::::: ::
;; BANCO 0: Inicio del programa y de la ROM
;; :::::::::::::::::::::::::::::::::::::::::::::::::::::: ::
.bank 0
.org $8000

RESET:
SEI    ;; Desactivar las peticiones de interrupcion (IRQ)
CLD    ;; No vamos a usar el modo decimal. Se desactiva
LDX #$40 ;; X = 0x40
STX $4017 ;; Desactivar las peticiones de interrupcion (IRQ)
        ;; del contador de frames de la APU
LDX #$FF ;; X = 0xFF
TXS    ;; Preparamos la pila para su uso posterior
INX    ;; X = 0x00
STX $2000 ;; Desactivar, por ahora, la interrupcion NMI
STX $2001 ;; Estableciendo los bits 3 y 4 a un valor de 0,
        ;; desactivar por ahora el dibujado de sprites
        ;; en cuanto a personajes y a fondo
STX $4010 ;; Estableciendo el bit 7 a un valor de 0,
        ;; se desactivan las peticiones de interrupcion (IRQ)
        ;; del "delta modulation channel" (DMC), relacionado
        ;; con la Unidad de Procesamiento de Audio (APU)
```

```

    BIT $2002
vblankwait1:
    BIT $2002
    BPL vblankwait1

;; Esta es la rutina de limpieza de la memoria de la RAM
;; Borra toda la memoria que hubiera y la establece a 0
;; En cuanto a la zona de sprites, la memoria la establece a 0xFE
;; ya que esto corresponde a un indice no valido de sprite
clrmem:
    LDA #$00
    STA $0000, x
    STA $0100, x
    STA $0300, x
    STA $0400, x
    STA $0500, x
    STA $0600, x
    STA $0700, x
    LDA #$FE
    STA $0200, x
    INX
    BNE clrmem

vblankwait2:
    BIT $2002
    BPL vblankwait2

;; A partir de aqui la PPU esta totalmente lista para utilizar
;; ..

```

Finalmente se puede ver la escritura en dos registros de memoria donde, para más información, se puede consultar en la tabla que está en el apartado B.2.

6.2.3. Rutinas de inicio del juego y bucle principal

Realmente no hay que escribir nada nuevo en esta zona, por ahora, puesto que las dos rutinas escritas en 6.1 pertenecen a esta zona. Estas dos y todas aquellas que se escriban en el futuro servirán para inicializar los datos de nuestro juego.

Tal y como se ve en 6.4, desde la etiqueta `GameEngine` hasta la `GameEngine_END` se llevará a cabo el bucle infinito que irá actualizando nuestro juego.

Código 6.4: Bucle principal del juego

```

;; ...
;; END: Rutina_LOAD SPRITES

;; Queremos que se active la interrupcion NMI
;; Ademas de elegir los sprites de la patter table 0
;; y el fondo de la pattern table 1
LDA #%10010000
STA $2000

;; Activamos el dibujado de sprites, el fondo y desactivamos el
;; clipping de los sprites en la parte izquierda de la pantalla
LDA #%00011110
STA $2001

;; Bucle infinito
;; El codigo del juego va a partir de la etiqueta NMI

```



```

Forever:
    JMP Forever

;; La etiqueta NMI es llamada cada frame gracias al
;; refresco continuo de la pantalla
NMI:
    ;; Transferencia de la memoria de sprites a la PPU
    ;; mediante el uso de DMA (Direct Memory Access)
    LDA #$00
    STA $2003
    LDA #$02
    STA $4014

    ;; A continuacion vendria la lectura de los controles

GameEngine:
    ;; Flujo real del juego

GameEngine_END:

    LDA #%10010000
    STA $2000
    LDA #%00011110
    STA $2001

    ;; Por ahora no hay scroll
    LDA #$00
    STA $2005
    STA $2005

    RTI
;; ...

```

Debo hacer un pequeño apunte sobre los últimos valores que son almacenados en la PPU desde los registros 0x2000 y 0x2001: Tal y como se ve, se propone un modo para poder visualizar los gráficos. Es importante saber, que estas dos llamadas deben ser las últimas en ser llamadas, sino dará problemas a la hora de ejecutar distintas rutinas de nuestro código aunque estén bien escritas.

Un ejemplo es a la hora de la carga del mapa, que se ve más adelante, si se activa, por ejemplo, la interrupción NMI, como se ve en el código, con el registro 0x2000, causa que la PPU no funcione con normalidad, tienda al reinicio de su contador interno y por lo tanto el mapa sea cargado en memoria de forma incorrecta.

6.2.4. Subrutinas del juego y zona de datos

Tal y como indica el nombre de la sección, aquí se establecen unas subrutinas que son llamadas durante la ejecución del bucle infinito del juego. Se debe hacer de esta forma para establecer una limpieza y no tener todas las funciones incrustadas dentro de dos etiquetas.

A continuación, vendría aquella zona donde almacenamos las variables necesarias para el correcto funcionamiento del juego. Aunque cuidado, porque los valores que se definan en esta zona, no podrán ser modificados durante la ejecución del programa debido a que las estamos almacenando en la ROM, al igual que todo lo anterior. Destaca al inicio del fragmento de código 6.5 la reutilización de las macros `.bank` y `.org`. Los detalles sobre el funcionamiento

de los bancos se encuentra en el anexo B.9.

Código 6.5: Subrutinas y datos

```
;; ...
;; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; BANCO 1: 8Kb de espacio
;; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
.bank 1
.org $A000

DIR_PALETA_COLORES:
;; Fondo
.db $22,$29,$1A,$0F, $22,$36,$17,$0F, $22,$30,$21,$0F, $22,$27,$17,$0F
;; Sprites
.db $22,$16,$27,$18, $22,$02,$38,$3C, $22,$1C,$15,$14, $22,$02,$38,$3C

DIR_SPRITES:
.db $80, $32, $00, $80 ;; Sprite 0
.db $80, $33, $00, $88 ;; Sprite 1
.db $88, $34, $00, $80 ;; Sprite 2
.db $88, $35, $00, $88 ;; Sprite 3

;; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; BANCO 2: 8Kb de espacio
;; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
.bank 2
.org $C000
;; Mas espacio para datos
;; ..
```

6.2.5. Zona de vectores de interrupción y sprites

Finalmente debemos incluir en nuestro archivo las etiquetas a las que saltará el procesador cuando ocurra una de las tres interrupciones posibles. Es importante saber que la zona de los vectores de interrupción no puede cambiar de dirección de memoria: Siempre comienza en 0xFFFFA y ocupa 6 bytes.

Código 6.6: Vectores de interrupción y adición de sprites

```
;; ...
;; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; BANCO 3: 8Kb de espacio
;; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
.bank 3
.org $E000
;; Comienzo de los tres vectores de interrupcion
;; vector = direccion de memoria = 2 bytes
.org $FFFA

;; En el momento que ocurra una interrupcion de tipo NMI
;; el procesador saltara a la etiqueta RESET
.dw NMI

;; Cuando el procesador se inicie o se reinicie
;; saltara a la etiqueta RESET
.dw RESET

;; Cuando ocurra una interrupcion externa de tipo IRQ
;; saltara a la siguiente etiqueta
.dw 0
```

```
;; Incluir los 8Kb de sprites
.bank 4
.org $0000
.incbin "mario.chr"
;; -FIN DEL ARCHIVO-
```

Al ver el fragmento de código 6.6 puedes preguntarte el porqué de la macro `.incbin` y ¿pero no estoy sobreescribiendo parte de la memoria RAM? Pues no, porque sólo se está incluyendo un binario: Los sprites que queremos añadir a nuestro juego.

6.3. Compilando el primer programa

Una vez ya se tiene todo lo anterior en cuenta, es hora de ver todo lo que se ha hecho compilando el programa: Para ello se necesita un compilador para NES. En mi caso estoy utilizando NESASM3.

Código 6.7: Como compilar un juego para NES

```
Estas son las opciones que trae el compilador
> NEASM3.exe --help

NES Assembler (v3.1)

NESASM3 [-options] [-? (for help)] infile
-s/S : show segment usage
-l # : listing file output level (0-3)
-m : force macro expansion in listing
-raw : prevent adding a ROM header
-srec : create a Motorola S-record file
infile : file to be assembled

Un ejemplo de compilación sería el siguiente
> NESASM3.exe -S game.asm
```

En cuanto se compile el juego ya podemos cargarlo en cualquier emulador de NES, que en mi caso utilizo fceux debido a que nos permite ver la memoria de programa en todo momento y, lo más importante, tiene un debugger que nos sacará de un apuro en caso de que algo falle. En todo caso, lo mejor siempre es utilizar varios emuladores para realizar las pruebas y si es posible contar con el hardware original.

6.4. Movimiento del personaje principal

Una vez que tenemos nuestro personaje principal pintado, para hacer que se pueda desplazarse por la pantalla primero hay que activar la lectura de los controles, que funciona mediante el acceso al registro `0x4016`. En el anexo B.5 se puede encontrar más información de como funciona esta rutina, además de que es importante saber que **hay un orden concreto** a la hora de leer tanto las pulsaciones del jugador 1 como las del jugador 2. Esta rutina se puede realizar perfectamente tal y como se explica en el anexo anteriormente nombrado, aunque una rutina tan larga llegue a confusión.

Por ello, una de las opciones a las que he podido llegar ha sido mediante la utilización de una tabla de saltos tal y como se ve en el fragmento de código 6.8: Ésta funciona mediante



Figura 6.3: ¡Hola mundo!

la obtención de la función correspondiente gracias al byte que devuelve la lectura desde el registro del controlador.

Código 6.8: Tabla de saltos utilizada para el movimiento

```
DIR_CONTROLLER_ATTACKING_TABLE:
    .dw SRUTINA_Press_A
    .dw SRUTINA_Press_B
    .dw SRUTINA_Press_Select
    .dw SRUTINA_Press_Start
    .dw SRUTINA_Press_Up
    .dw SRUTINA_Press_Down
    .dw SRUTINA_Press_Left
    .dw SRUTINA_Press_Right
```

6.5. Pintado del fondo

El pintado del fondo se realiza de la misma forma de la que se cargaban las paletas de colores en memoria, para ello se sigue esquema muy parecido donde solo cambia la dirección que atacamos en la PPU.

Código 6.9: Pintado del fondo no optimizado

```
:: ...
;; Queremos modificar la primera name table, que se encuentra
;; en la direccion 0x2000 de la PPU
LDA $2002
LDA #$20
LDA $2006
LDA #$00
STA $2006

LDA #$24
STA $2007

LDA #$24
STA $2007

LDA #$24
```

```
STA $2007
```

```
; ...
```

Si se observa el fragmento de código 6.9, teniendo en cuenta que una pantalla entera ocupa 1024 bytes, se llega a la conclusión que esa no es la forma correcta de realizar la carga de nuestro fondo. Para ello, se va a utilizar un método parecido a los utilizados anteriormente, además que si en un futuro se quisiera cambiar el mapa, sería algo bastante tedioso.

Por lo tanto, se suele construir una rutina que cargue el mapa en un bucle. Ésta, gracias al acceso de la memoria mediante el modo indirecto post-indexado (A.3.8), recorreremos la información que está guardada en nuestra ROM, a la que, finalmente, se traslada a la PPU utilizando el registro correspondiente para ello, sin olvidar establecer la dirección de memoria que actuará como destino en la PPU.

Una vez se ha escrito dicha rutina, la figura 6.4 es un espejo de lo que sería la memoria de la PPU, en ese caso, de la primera nametable.

```
002000: 24 24 24 24 24 24 24 24 24 24 24 24 24 24 24 24
002010: 24 24 24 24 24 24 24 24 24 24 24 24 24 24 24 24
002020: 24 24 24 24 24 24 24 24 24 24 24 24 24 24 24 24
002030: 24 24 24 24 24 24 24 24 24 24 24 24 24 24 24 24
002040: 24 24 24 24 45 45 24 24 45 45 45 45 45 45 24 24
002050: 24 24 24 24 24 24 24 24 24 24 24 24 53 54 24 24
002060: 24 24 24 24 47 47 24 24 47 47 47 47 47 47 24 24
002070: 24 24 24 24 24 24 24 24 24 24 24 24 55 56 24 24
002080: 24 24 24 24 24 24 24 24 24 24 24 24 24 24 24 24
... ..
```

Figura 6.4: Representación del mapa en memoria

Finalmente, si se ha hecho bien la transferencia de datos hacia la PPU y ejecutamos el juego, se puede ver algo parecido a lo que observamos en la imagen 6.5, aunque si nos fijamos, se observa que los colores aún no están del todo bien.



Figura 6.5: Primer pintado del fondo

Por lo tanto, se va a seguir un proceso **idéntico** al seguido en los procesos de transferencia de memoria de la CPU hacia la PPU y rellenar sus attribute tables, aunque con un pequeño matiz relacionado con el cómo la NES trata las paletas de colores, que se puede revisar en el anexo C.3. Así que, finalmente, en la figura 6.6, se puede ver el resultado final.



Figura 6.6: Primer pintado correcto del fondo

6.6. Implementación del scroll

Hasta ahora, el videojuego solo tenía una pantalla fija, sin scroll alguno. Eso es debido a las líneas que podemos ver en el fragmento de código 6.10, que, básicamente, evitaba que esto pasase debido a que el registro 0x2005 es el que controla el desplazamiento del scroll y en cada entrada al bucle principal del juego se estaba cargando un valor de 0, tanto para el scroll horizontal, en el primer acceso, como para el scroll vertical, en el segundo acceso. No obstante, puesto que solo se va a implementar un scroll horizontal, la segunda escritura en el registro 0x2005 se va a mantener con el mismo valor.

Código 6.10: Instrucciones que inhabilitan el scroll

```
LDA #$00  
STA $2005  
STA $2005
```

Por lo tanto, para habilitar el scroll, o desplazamiento de la pantalla, hay que cambiar el valor con el que se accede la primera vez al registro 0x2005, ya que es la que corresponde al scroll en el eje horizontal, mientras que el segundo acceso corresponde al eje vertical.

Para hacer una pequeña prueba de como funciona el scroll, lo mejor es establecer una variable que incremente en el momento que se pulsa un botón. Como no, este valor, como se ha dicho justo en el párrafo de antes, debe ser el valor con el que se acceda la primera vez al registro 0x2005.

En la figura 6.8 se muestran las dos primeras nametables: La primera representa la pantalla cuando el valor de scroll en horizontal es 0, en cambio la segunda es cuando este valor es 16. Por lo tanto, **el scroll es, el desplazamiento entre las nametables que escojamos**, en este caso es horizontal al utilizar las dos primeras. Si con lo que se tiene por ahora se llega al final de la pantalla, se puede ver que se vuelve al principio de todo: Esto sucede porque el

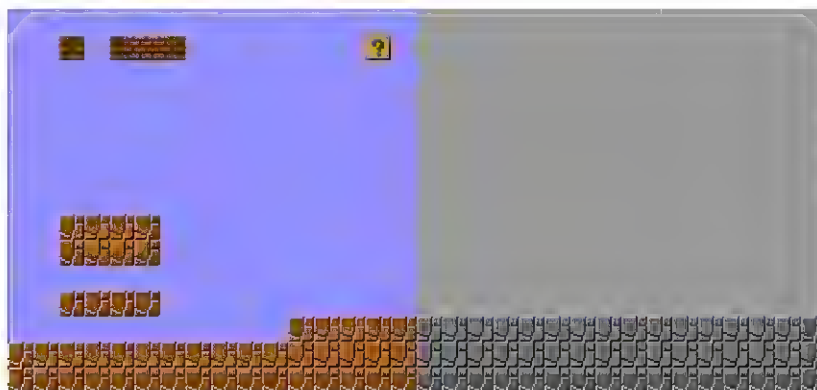


Figura 6.7: SCROLL = 0



Figura 6.8: SCROLL = 16

registro 0x2000 tiene asignado que solo utilice la nametable numero 0, por lo que se soluciona estableciendo el bit concreto a 1 y viceversa cuando esté utilizando la nametable 1 y quiera pasar a la 0.

Ahora que ya hay un scroll medianamente funcional, puesto que el principal objetivo de esta funcionalidad es el hecho de no tener una pantalla parada e inmóvil y tener un mapa grande por el que nuestro personaje pueda moverse con cierta libertad, a continuación se va a introducir el sistema de construcción del mapa para que, a medida que nos movemos por la pantalla se vayan dibujando, dinámicamente, columnas nuevas. Aunque, para ello, primeramente hay que almacenar el mapa respecto a las columnas y no respecto a las filas tal y como se muestran las figuras 6.9a y 6.9b: De esta forma podremos avanzar en el dibujado del mapa según las columnas y no las filas, puesto que se va a utilizar un scroll horizontal.

Ahora sí, para implementar la rutina de dibujado de nuevas columnas se van a seguir tres pasos: **Cuando** dibujar, **donde** dibujar y **cómo** dibujar.

1. **CUANDO:** Cuando el contador del scroll, el que se utiliza para realizar la primera escritura en el registro 0x2005 como se ha visto antes, sea múltiplo de 8.
2. **DONDE:** Por una parte hay que obtener la dirección de memoria de la dirección de destino que tenemos que sobrescribir. Para ello, gracias a la variable que se va



Diagrama que muestra un mapa de Super Mario Bros. con un recuadro que representa el almacenamiento de la información del mapa por filas. El recuadro contiene una matriz de 5x5 con los siguientes valores:

24	24	24	24	24
24	24	24	24	24
24	24	24	24	45
24	24	24	24	47
24	24	24	24	24

(a) Almacenamiento del mapa en cuanto a filas

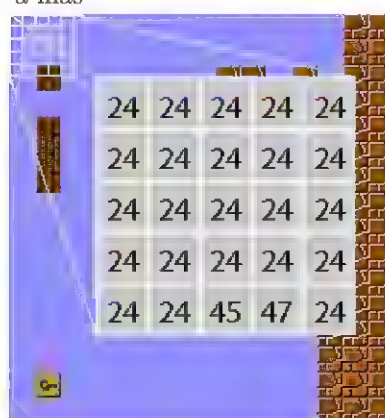


Diagrama que muestra el mismo mapa de Super Mario Bros. con un recuadro que representa el almacenamiento de la información del mapa por columnas. El recuadro contiene una matriz de 5x5 con los siguientes valores:

24	24	24	24	24
24	24	24	24	24
24	24	24	24	24
24	24	24	24	24
24	24	45	47	24

(b) Almacenamiento del mapa en cuanto a columnas

Figura 6.9: Formas de almacenar la información del mapa

incrementando, `CTR_SCROLL` por ejemplo, se obtiene el byte bajo: Esta variable se divide entre 8 realizando un total de tres rotaciones de bits hacia la derecha resultando en un valor entre 0 y 31, indicando la fila que hay que dibujar en concreto.

Seguidamente, el byte alto se consigue gracias a aquella variable que se utiliza para indicar que nametable tenemos activa en pantalla. No obstante, hay que recalcar que **se dibuja siempre en la nametable que no se utiliza**. En las tablas 6.1 se pueden ver los distintos resultados que se obtienen dependiendo del byte que se pase como entrada.

Por otra parte hay que obtener la dirección de memoria de dónde tenemos que empezar a coger la información almacenada del mapa. Para estos cálculos se utiliza aquella variable que se utiliza para contar las columnas que se van pintando, es decir, en el caso que se haya pintado una pantalla entera, este valor será de 32 en decimal o 0x20 en hexadecimal.

El byte bajo se consigue guardando los tres bits más bajos de esta variable, mientras que para el byte alto se guardan los cinco bits más altos, todo ello mediante operaciones

Obtención del byte bajo		Obtención del byte alto	
CTR_SCROLL	Byte bajo	NAMETABLE	Byte alto
00	00	00	20
08	01	01	24
10	02		
18	03		
20	04		
...	...		
E0	1C		
E8	1C		
F0	1E		
F8	1F		

Tabla 6.1: Obtención de la dirección de memoria destino para dibujar

de rotación de bits. En las tablas 6.2 se pueden ver estos resultados, al igual que las anteriores, la salida depende del byte que se le pase como entrada.

Obtención del byte bajo		Obtención del byte alto	
COLUMNA	Byte bajo	COLUMNA	Byte alto
_0	00	00 - 07	00
_1	20	08 - 0F	01
_2	40	10 - 17	02
_3	60	18 - 1F	03
_4	80	20 - 27	04
_5	A0	28 - 2F	05
_6	C0	30 - 37	06
_7	E0	38 - 3F	07
	

Tabla 6.2: Obtención de la dirección de memoria de origen para dibujar

Como se puede observar de los resultados de las tablas anteriores (6.2), se puede sacar en claro que cada pantalla puede ocupar un total de 1024 bytes y que cada columna ocupa 32 bytes.

3. **COMO:** En esta última fase hay que utilizar todos aquellos datos que se han obtenido anteriormente. Para ello se va a utilizar la dirección de memoria destino para acceder a la PPU y la dirección de memoria origen para saber desde donde conseguir los datos que se encuentran almacenados en la memoria.

Hay que tener en cuenta que cada columna nueva se va a pintar siempre que se haya incrementado 8 veces la variable con la que controlamos el scroll y activar el modo de incremento de 32 de la PPU. Un ejemplo del algoritmo se puede observar en el pseudocódigo 1.

Algorithm 1 Algoritmo de pintado de nuevas columnas

Require: $S \& 7 == 0$

$getDireccionDestino(S, N) \rightarrow T$

$getDireccionOrigen(C) \rightarrow F$

$F+ = MAPA$

$preparaPPU()$

$almacenaTEnPPU(T)$

$pinta30Bytes(F)$

6.6.1. Aplicando los colores correctamente

En el momento que se pruebe el juego con solo el sistema de scroll se puede ver claramente que los colores con los que pintados los distintos sprites del fondo pueden no ser los correctos a los que se quería usar en un principio. Eso es debido a que este sistema solo cambia los datos de la apariencia, por así decirlo, y no se realiza una escritura de nuevos datos sobre las attribute tables, es decir, que paleta utiliza cada zona de la pantalla.

Para saber la forma en la que hay que almacenar esta información, primeramente hay que saber como está organizada esa zona: En la figura 6.10 se muestra la attribute table que está asociada a la primera pattern table, empezando en la dirección de memoria 0x23C0. Se podría decir que 8 bytes almacenados en memoria es igual a sobrepasar una fila completa de sprites de la pantalla.

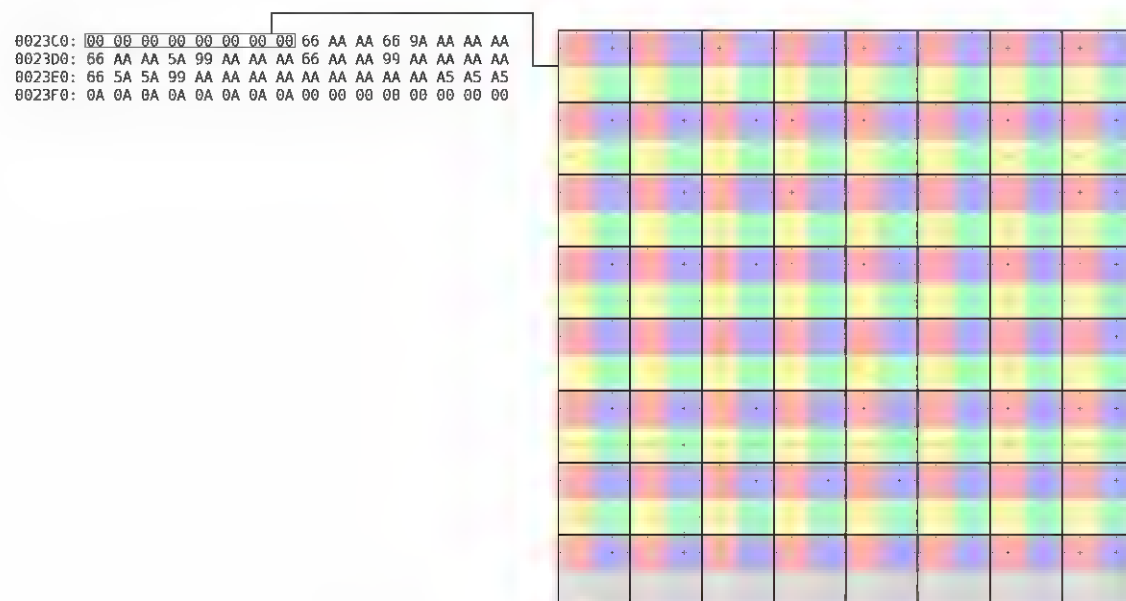


Figura 6.10: Organización de la attribute table 0

Al igual que el sistema de scroll se van a seguir los mismos tres pasos:

1. **CUANDO:** Esta rutina tendrá los mismos principios que el sistema de scroll, por lo

que las operaciones serán parecidas. Uno de los cambios más importantes es el hecho que no debemos de escribir nuevos atributos cada 8 incrementos de la variable del scroll, **sino cada 32**, ya que, como se ha observado en la figura 6.10, cada byte "colorea" una ventana de 4x4 sprites.

2. **DONDE:** Para obtener las direcciones de memoria necesarias se necesitan tres variables al entrar a la rutina: **La variable auxiliar de scroll, que nametable se esta utilizando** y por último **el número de columnas dibujadas**.

La dirección de memoria destino se consigue a través de la variable auxiliar de scroll y la nametable que se está usando en el momento. Concretamente, como se muestran en las tablas 6.3, con la primera se obtiene el byte bajo y con la segunda el byte alto.

Obtención del byte bajo		Obtención del byte alto	
CTR_SCROLL	Byte bajo	NAMETABLE	Byte alto
00	00 + C0	00	00 + 23
20	01 + C0	01	04 + 23
40	02 + C0		
60	03 + C0		
80	04 + C0		
A0	05 + C0		
C0	06 + C0		
E0	07 + C0		

Tabla 6.3: Obtención de la dirección de memoria destino para escribir los nuevos atributos

Por otra parte, la dirección de memoria origen se consigue a través del número de columnas que ya han sido dibujadas como muestran las tablas 6.4.

Obtención del byte bajo		Obtención del byte alto	
Columnas	Offset byte bajo	Columnas	Offset byte alto
00 - 03	00	00 - 7F	00
04 - 07	08	80 - FF	01
08 - 0B	10		
0C - 0F	18		
10 - 13	20		
...	...		

Tabla 6.4: Obtención de la dirección de memoria origen para obtener los nuevos atributos

CUIDADO! Hay que tener en cuenta el flag carry al sumar los bytes bajo y alto de la dirección de memoria donde se encuentran los atributos almacenados a los offset calculados anteriormente.

3. **COMO:** Al tener que almacenarla de la misma forma que los datos del nivel, es decir,

verticalmente, como se observa en la figura 6.11, cada vez que se escriba algún byte habrá que sumar 8 a la dirección de memoria destino para apuntar al bloque de debajo, y no al de la derecha.

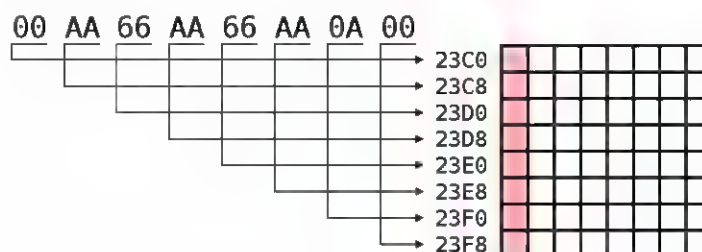


Figura 6.11: Forma de almacenar la información de las attribute table

6.7. Colisiones

En caso que el videojuego que se vaya a desarrollar trate de una nave o algún vehículo que vuela, no habría que preocuparse mucho por las colisiones con el entorno, aunque depende del tipo de juego que se vaya a hacer. En mi caso, puesto que es un plataformas, quedaría bastante raro que el personaje volara por en medio de la pantalla mientras hace caso omiso de los obstáculos que están dispuestos por el mapa. Por ello, esta sección está dedicada a la explicación del sistema de colisiones implementado.

Como se ve en la figura 6.12, básicamente se divide el mapa en ceros y unos, siendo 1 los bloques en color, con los que se puede colisionar, y, por ende, 0 con los que no se puede colisionar.



Figura 6.12: Bloques con colisiones en color

Para ello, la forma que he elegido de almacenarlo en memoria es la siguiente: Por cada sprite que hay en pantalla en la coordenada X, hay un bit que se corresponde con ésta. En cambio, con la coordenada Y por cada dos sprites en pantalla se corresponde un bit en el

mapa de colisiones. Para una mejor explicación, se puede observar la figura 6.13: En ella aparece un recorte de la figura que se ha mostrado anteriormente con las agrupaciones de los sprites según he explicado justo antes.

La figura de la derecha indica como quedaría en memoria finalmente.

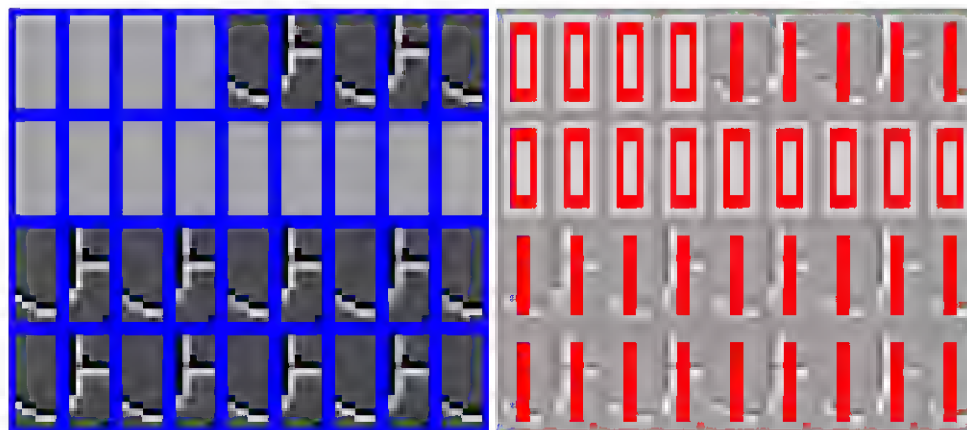


Figura 6.13: Esquema de almacenamiento del mapa de colisiones en memoria

En concreto he elegido esta forma de representar el mapa de colisiones por dos motivos:

1. Almacenamiento en memoria

Aún habiendo algunas formas más de almacenar este mapa de colisiones, de esta forma, una pantalla entera solo ocupa un total de 64 bytes. Podría ocupar menos, en concreto 32 bytes, pero eso me quitaría de una pequeña holgura a la hora de calcular las colisiones en la coordenada X y también podría ocupar más, relacionando 1 bit con cada sprite en pantalla, llegando a 128 bytes.

2. Rapidez de cálculo

Aunque no lo he comentado en los apartados anteriores, las colisiones se pueden calcular directamente en el mapa que está almacenado en memoria, aunque se necesite de algunos medios adicionales para ello, como guardar los índices de los sprites, que corresponden a las pattern table, que colisionan y los que no. De esta forma, solo hay que hacer unos cálculos, que se explicarán a continuación y finalmente una operación AND con el byte correspondiente.

6.7.1. Cómo calcular las colisiones

Para realizar el cálculo de las colisiones se necesitan las coordenadas X e Y que queremos comprobar. En nuestro caso, pueden ser las coordenadas en las que se encuentra el personaje o las siguientes, hacia la derecha, cuando éste se mueva. Por ello, voy a dividir su cálculo en tres partes: **El cálculo del byte en X, el cálculo del bit que se va a atacar y el cálculo del byte en Y.**

1. Cálculo del byte X

Para el cálculo del byte en X se necesita, como no, la coordenada en X a comprobar. Puesto que he decidido que el mapa de colisiones va a tener un ancho de 4 bytes, como he explicado antes, nos quedaremos con los bits 6 y 7 de la coordenada X: Éstos indican en que rango de la pantalla nos encontramos, tal y como se indica en la figura 6.14.



Figura 6.14: Cálculo del byte X para las colisiones

2. Cálculo del bit a atacar

Para saber el bit al que hay que atacar en el mapa de colisiones, necesitamos realizar un cálculo similar al anterior para descubrir el byte al que pertenecemos: Para ello nos tenemos que quedar con los bits 4 y 5 de la coordenada en X. Gracias a ellos podemos saber en que subzona estamos, dentro de la zona que hemos calculado en el paso 1. En la figura 6.15 se puede observar un esquema de este concepto.

Estos dos bits se utilizan para conseguir, a través de una tabla almacenada en memoria, el bit concreto al que atacar al final de los cálculos. Estos datos pueden verse en la tabla 6.5, donde, según la zona en la que estemos un byte que solo tiene un bit activo.

Bits	Byte devuelto
00	10000000
01	00100000
10	00001000
11	00000010

Tabla 6.5: Tabla que almacena el bit a atacar según la coordenada X

Pero hay más, puesto que hay que tener en cuenta que cada bit en X corresponde a



Figura 6.15: Cálculo de la subzona según la coordenada X

un sprite distinto. Por lo tanto, en el caso que el bit 3 de la coordenada X esté activo, se realiza una operación de rotación de bits hacia la derecha sobre el byte que se ha conseguido de la tabla anterior. Este bit indica que el personaje está en el primer sprite que pertenece al bloque, o en el segundo.

3. Cálculo del byte Y

Por último hay que determinar en qué fila nos encontramos: Para ello se necesitan los cuatro bits más altos de la coordenada Y, ya que cada bit corresponde a dos bloques en esta coordenada, como se ve en el esquema de la figura 6.16. En concreto, este valor, multiplicado por cuatro será el valor que nos proporcione el desplazamiento en Y dentro del mapa de memoria.

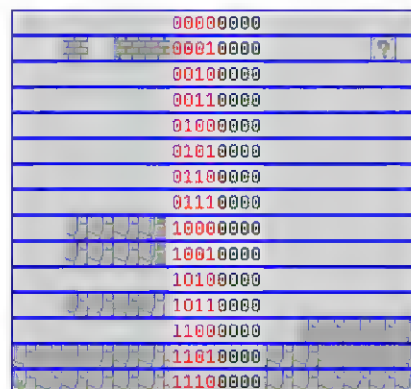


Figura 6.16: Cálculo del desplazamiento en Y del mapa de colisiones

Finalmente solo hay que realizar una operación AND, utilizando el byte conseguido en el paso dos, con el byte que corresponde a la siguiente fórmula:

$$\text{byte_final} = \text{byte_y} * 4 + \text{byte_x}$$

6.7.2. Integración de las colisiones con el scroll

Una vez están las colisiones implementadas para una **pantalla estática**, toca cambiar un par de cosas para que este sistema pueda integrarse con el sistema de scroll que se explica en 6.6.

Para ello, debe cambiarse la forma en la que se almacena el mapa de colisiones en memoria, puesto que ahora está almacenado, por así decirlo, tal cual lo vemos en pantalla. De esta forma, no podemos indicarle un offset para que, a medida que se haga scroll, se añada un offset y el mapa de colisiones se desplace. Por ello, se va a cambiar la orientación de almacenamiento del mapa y, por consecuencia, **todos los cálculos asociados a la coordenada X serán ahora de la coordenada Y y viceversa**.

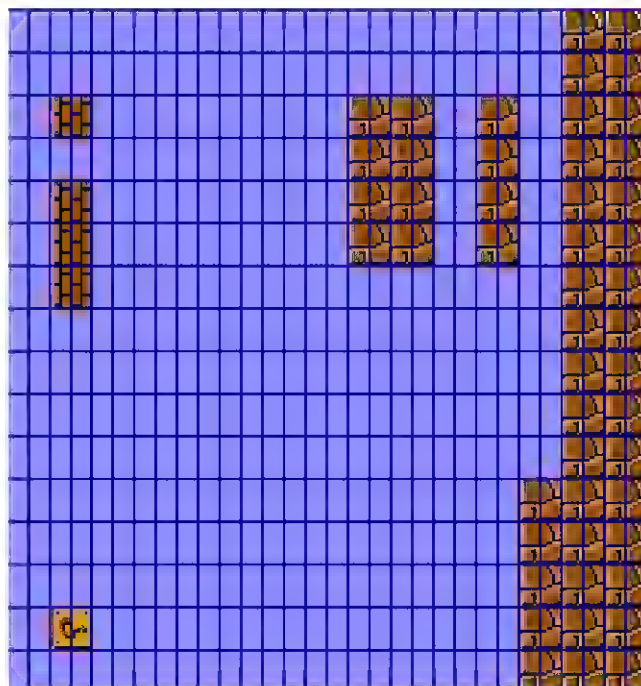


Figura 6.17: Almacenamiento final del mapa en memoria

Como se observa en la figura 6.17 al final es solamente voltear el mapa de colisiones y aplicarle un efecto espejo para que funcione todo bien. Finalmente, para terminar dicha integración, hay que definir un offset que aumente cada dos veces que pintemos una columna en el scroll. Añadiendo los respectivos cambios a la función anterior y añadiendo el offset, dicha función queda tal que así:

$$\text{byte_final} = (\text{offset} + \text{byte_x}) * 4 + \text{byte_y}$$

Ahora bien, de esta forma las colisiones son un poco cuadradas, puesto que el valor del offset siempre va a tener en cuenta cuán de lejos se llega con el scroll y nunca la posición que estamos comprobando en la rutina de las colisiones. Por lo tanto se contemplan tres situaciones en las que tenemos que cambiar el valor de ese offset según indica la figura 6.18.

(a) **Bloque entero**

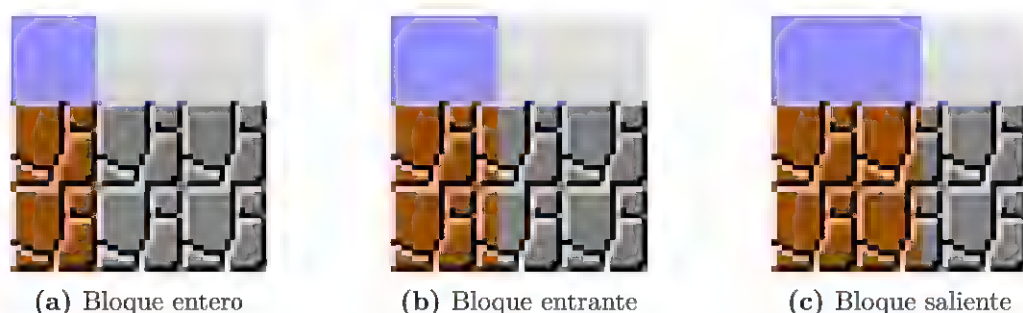


Figura 6.18: Posibles situaciones de los bloques según el scroll

Trataremos con un bloque entero aquel que cumple a la perfección el esquema que se ha seguido hasta ahora, es decir, aquel que no sea cortado por la parte derecha de la pantalla. En este caso **el offset permanecerá intacto**.

(b) Bloque entrante

Trataremos con un bloque entrante en el momento que la parte derecha de la pantalla corte el bloque en **su primera mitad**, hasta el píxel 7, contando éste último. Por ello al valor de offset hay que sumarle los píxeles que entran:

```
offset += píxeles_entrantes
```

(c) Bloque saliente

Trataremos con un bloque saliente en el momento que la parte derecha de la pantalla corte el bloque en **su segunda mitad**, hasta su último píxel. Por ello al valor de offset hay que restarle los píxeles que entran:

```
offset -= píxeles_entrantes
```

6.8. Enemigos

En un videojuego de plataformas, los enemigos son casi obligatorios, por lo tanto, en esta sección se va a explicar el sistema que se ha usado para crear y actualizar los distintos enemigos que se vayan creando para interactuar con el personaje principal y los distintos elementos que conforman cada nivel. Puesto que existe tiene un sistema de scroll que dibuja nuevas columnas, dentro de esa rutina se va a llamar a otra que comprueba si hay que crear nuevos enemigos.

En un principio tenía pensado hacer otro mapa, como el que se ha hecho para las colisiones, pero no es nada rentable por dos razones:

1. Los nuevos enemigos son creados **siempre** en el lado derecho de la pantalla, solo habría que cambiar la altura en la que aparecen.
2. Da igual cuantas veces se inicie un nivel que los enemigos siempre serán los mismos para éste.

Por lo tanto, solo se necesitan dos valores para decir **cuando** y **donde** debe aparecer un enemigo en el nivel: El primero de ellos será la columna absoluta de todo el nivel en la que debe aparecer y el segundo su altura, no obstante se podría utilizar un tercer valor que indique la X en la que debe aparecer. Cabe recalcar que la columna elegida es la numerada desde el principio del nivel, es decir, la primera columna que se dibuja de todo el nivel es la 0, como se indica en la figura 6.19, y las que están marcadas son las elegidas para crear un enemigo en ellas.

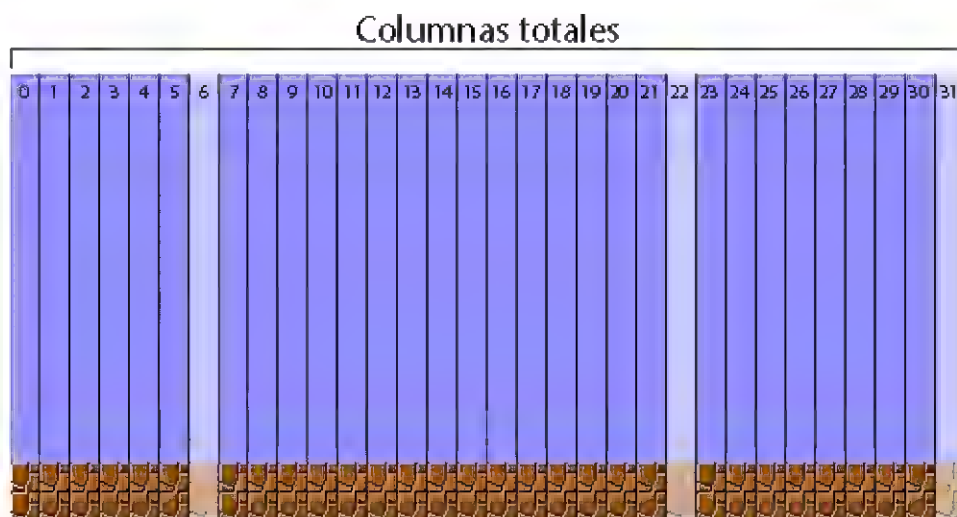


Figura 6.19: Columnas seleccionadas para la aparición de enemigos

Para el movimiento de los enemigos, se va a combinar dos tipos de desplazamientos: Un desplazamiento respecto a todo el nivel que se suma en el momento que el mapa hace scroll y otro que se corresponde con el movimiento propio de cada enemigo. Gracias a estas dos rutinas el usuario tendrá la sensación de que el enemigo vive en el nivel y no en la pantalla, como puede hacer nuestro personaje. Por supuesto, en el momento que el enemigo llega a la parte izquierda de la pantalla, éste es borrado.

Finalmente, hay que hacer que los enemigos puedan colisionar. Para ello se va a aplicar el mismo método con el que se ha hecho anteriormente con el héroe, pero con limitaciones: Puesto que no va a haber un solo enemigo siempre y pueden llegar a haber varios por pantalla a la vez, hay que limitar las veces que se comprueba su colisión, concretamente cada vez que recorra 16 píxeles, o, lo que es lo mismo, un bloque, ya que las colisiones son siempre enteras y nunca con bloques partidos.

6.9. Objetos y barra de estados

En el caso que se quieran introducir objetos por todo el nivel, como es mi caso, es algo sencillo y que, si se ha realizado el punto anterior sobre los enemigos, la rutina ya existe y solo hay que introducir un añadido a ésta.

El comportamiento de los objetos va a ser el siguiente:

1. No van a tener comportamiento propio, es decir, no se van a mover libremente por el mapa como pasa con los enemigos, aunque sí les afecta ese pequeño offset que se le aplica a éstos últimos para dar la sensación que pertenecen al propio nivel.
2. Los objetos podrán recogerse y se sumarán a una puntuación total.

En cuanto a la creación y la rutina de actualización de los objetos, éstas son idénticas a las de los enemigos, solo que los objetos no disponen de movimiento propio y que éstos pueden ser recogidos. Por otra parte, para que puedan ser recogidos, solo hay que comprobar las coordenadas X e Y tanto del objeto como del héroe y ver si están más cerca que una pequeña distancia que se ponga, como por ejemplo 6 píxeles. Finalmente el objeto se elimina de la memoria y se anota en una variable que se ha recogido un item.

Una vez ya se tiene la cuenta de los items que se han recogido en un nivel, se puede empezar a hacer una barra de estados, aunque primero hay que hablar del **sprite 0**:

El sprite 0, es aquel que ocupa las direcciones de memoria 0x0200 a 0x0203 y para la NES es un tanto especial por el hecho de que puede saber si un píxel no transparente de sí mismo se encuentra en las mismas coordenadas que un píxel no transparente del fondo. Para ello se comprueba el bit 6 de la dirección de memoria 0x2002 (B.2). Una vez comprobado solo hay que establecer un pequeño bucle en el que se compruebe que este bit se activa y a continuación seguir con las demás rutinas.

Por ello, en el momento que se activa, a la NES le ha dado suficiente tiempo para no actualizar el número de líneas de pantalla que el programador establezca. Debido a esto todo aquello que se pinte en esa zona no se moverá con el scroll que hay establecido en el sistema del juego, por lo que se puede indicar la puntuación actual del jugador pintando en el fondo y sin usar sprites.

CUIDADO! porque si en el juego esto no ocurre, el programa se quedará en un bucle infinito y nunca va a salir de él.

A continuación, hay que cambiar un poco la estructura de actualización del juego:

1. Para que se compruebe bien la colisión del sprite 0 antes hay que establecer las dos escrituras del scroll a las direcciones 0x2005 y 0x2006 a 0 además de elegir la nametable 0. Una vez ha colisionado, ya se podrán establecer los valores normales de scroll y la elección de nametable que se estaban usando hasta ahora.
2. Puesto que la rutina de dibujado de nuevas columnas afectaría al dibujado de la barra de estado, hay que añadir unos offset a distintas variables calculadas dentro de la propia rutina, ya que hay que indicar que no queremos pintar en las primeras filas de la pantalla, además de cambiar las veces que recorre el bucle de dibujado y desde donde empieza éste.

Finalmente solo falta pintar dicha zona de pantalla y establecer el comportamiento de la puntuación. En mi caso voy a establecer un numero fijo de objetos que se pueden recoger en cada nivel, por lo que, como se ve en la figura 6.20, se van a dibujar sprites "huecos" que indican que hay que recoger objetos. Estos sprites son sustituidos en el momento que es

recogido un objeto, puesto que se sabe con exactitud la dirección de memoria que corresponde cada uno.



Figura 6.20: Mostrando la puntuación en la barra de estado

6.10. Bank switching para construir una pantalla inicial

Hasta ahora se estaba utilizando el mapper número 0, que proporciona 32KB de memoria para la ROM en la CPU, 8KB para los sprites en la PPU y, algo muy importante, no proporciona la posibilidad del cambio de bancos de memoria o *bank swapping*, por lo que no se puede utilizar más información de la que se establece en un principio en memoria.

La problemática está en que no puedes establecer otra pattern table distinta para hacer una pantalla de título sin utilizar sprites que no pertenezcan al propio juego. Por ello, se pueden hacer dos cosas: O se utilizan los mismos sprites del juego para la pantalla del título o se utiliza otro mapper que deje realizar el bank swapping. En mi caso, he decidido utilizar otro mapper distinto, en concreto el mapper 3.

Aunque este mapper proporciona la misma cantidad de memoria para la ROM, a su vez proporciona un total de cuatro bancos de memoria para los sprites, por lo que se podrían hacer hasta un total de cuatro pattern tables totalmente distintas, aunque solo se van a utilizar dos.

Cada mapper tiene su forma distinta de realizar el cambio, en concreto para el mapper 3 se ha de incluir los datos en un banco nuevo justo debajo de donde estaban incluidos los datos para la pattern table antigua. Internamente no se utilizan estos números que se le indican a los bancos, sino que el primero que se incluya será el 0, el segundo el 1 y así sucesivamente. Para realizar el cambio de banco hay que escribir el número del banco que se quiera usar que se quiera usar tal y como se muestra en el fragmento de código 6.11, cargando en el registro A el banco que se desee usar.

Código 6.11: Rutina que realiza el bank swapping

```
BANK_SWITCH:
    TAX
    STA BANK_VALUES, X
    RTS

BANK_VALUES:
    .db $00, $01
```

Es **IMPORTANTE** saber que han de cambiarse los headers conforme a la memoria que estemos usando en cada momento, por lo que si se están utilizando dos bancos de memoria

para los sprites hay que cambiar el valor de `.ineschr`.

En la figura 6.21 se pueden ver los resultados. Además, también se realiza un cambio de paleta.

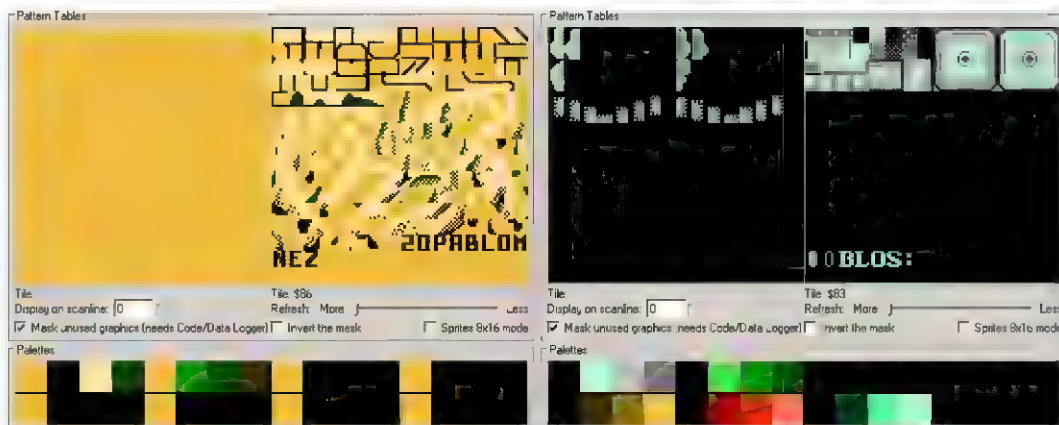


Figura 6.21: Pattern tables utilizadas

6.11. Compresión de los datos

Si se hacen un par de operaciones matemáticas, se saca en claro que cada pantalla puede llegar a ocupar un total de 1024 bytes en la ROM. En el caso que un nivel entero ocupe tan sólo 8 pantallas tendríamos 8192 bytes de la ROM ocupada, o, lo que es lo mismo, un banco entero. En el momento que esto ocurre hay que pensar en una forma de reducir el tamaño de la información que se almacena. Para ello existen varias soluciones entre las que se encuentran el uso de algoritmos que compriman la información, como Run-Length Encoding (RLE), el más común y del que existen diversas variantes, o la compresión de los sprites utilizados en metasprites.

En mi caso, me he decidido por el uso de metasprites ya que es un punto de partida por si en un futuro se decide aplicar algún tipo de algoritmo para comprimir aún más la información y porque la compresión que se consigue es siempre fija, teniendo en cuenta que cada nivel ocupe siempre lo mismo, ocupando la cuarta parte de la información original.

Para empezar, hay primero que definir los metasprites que van a ser utilizados en los niveles. Cada metasprite va a estar compuesto de 4 sprites tal y como se muestra en la figura 6.22. Cada color corresponde a un sprite, claro está en la imagen es algo simbólico, ya que un metasprite está definido por cada usuario y puede contener los 4 sprites que uno quiera. A la izquierda podría verse la representación final en pantalla mientras que a la derecha estaría como se guardaría la información que se almacenaría en la ROM para definir cada uno de los metasprites.

Una vez estén todos los metasprites definidos y en memoria, hay que tener en cuenta algunas cosas antes de seguir con otros aspectos del juego:

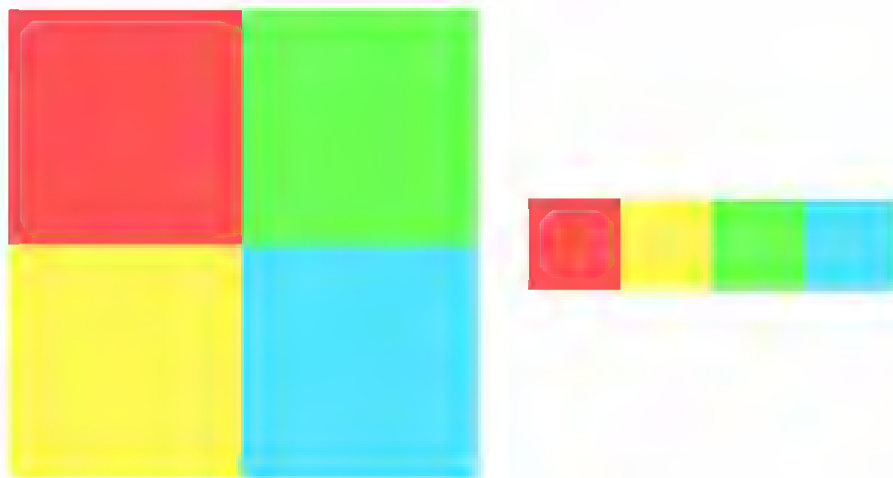


Figura 6.22: Organización de cada metasprite

1. Cambiar las rutinas de dibujado, ya que ahora se tiene que conseguir la información a partir de cada metasprite leído.
2. Llevar cuidado con los rangos al dibujar nuevas columnas: Ahora cada fila que se lea **ocupa la mitad del tamaño original**. Por el mismo motivo, una columna también ocupa la mitad.

6.12. Animaciones

En todo videojuego que se precie hay un diseño al que se le da mucha importancia por parte de sus creadores. A partir de este diseño es creado un vínculo que junta diversos aspectos del propio videojuego. Por ello, esta sección se va a centrar en uno de esos aspectos que dan vida a los personajes a través de un sistema de animaciones.

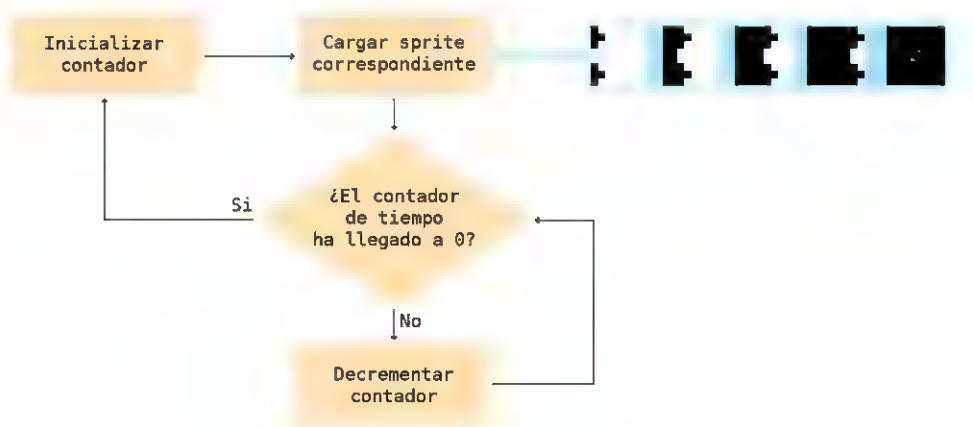


Figura 6.23: Diagrama con los pasos que sigue un sistema de animaciones

Como se observa en el diagrama representado en la figura 6.23, toda animación tiene, como

no, diversos frames: Cada uno de ellos es el estado en que se va a encontrar dicho sprite en un momento determinado, dependiendo siempre de la aquel que le preceda y el tipo de animación que se está usando.

Concretamente lo que se va a hacer es inicializar un **contador de animación**. Éste decrementará en cada frame y determina la velocidad de cambio entre los distintos sprites que componen la animación. Cuando menor sea el valor inicial del contador, mayor será la velocidad de la animación.

En el momento que llegue a cero, se vuelve a establecer el contador al valor inicial, para volver a repetir el proceso. A su vez, en ese mismo momento se incrementará un **contador interno** que indica el sprite que toca cargar ahora mismo, como se observa en la figura 6.24. Para obtener siempre los datos correctos es importante reiniciar el contador interno en el momento que llegue al último frame.



Figura 6.24: Numeración de los distintos frames de la animación

6.13. Seguimiento del proyecto

En esta sección se van a detallar los pasos que se han seguido desde que se ha comenzado el proyecto hasta su término.

6.13.1. Iteración 1 - Introducción al sistema NES

- **Fecha de inicio:** 7 de octubre de 2019
- **Fecha de finalización:** 17 de noviembre de 2019
- **Duración:** 42 días
- **Descripción:**

Esta es la primera iteración de todo el proyecto, por lo que he empezado con la escritura de esta memoria en \LaTeX primeramente con la creación de varios anexos relacionadas con los pequeños cerebros que componen la NES: La Unidad Central de Procesamiento (CPU) y la Unidad de Procesamiento de Imágenes (PPU).

Realmente no vengo con las manos vacías, puesto que unos meses antes de empezar el curso académico, en verano, empecé realizando un pequeño prototipo, a través de

una publicación en el foro de NintendoAge¹ que explica los principios básicos de como funciona la NES.

Ha acabado consistiendo en una pantalla estática en la que se controla a un personaje familiar, Mario, como placeholder para los futuros sprites, en la que hay unas colisiones básicas pero funcionales, como se muestra en la figura 6.25. Por lo que he realizado una versión mejorada de este código que ya escribí, ahora con mejor conocimiento del lenguaje ensamblador 6502.

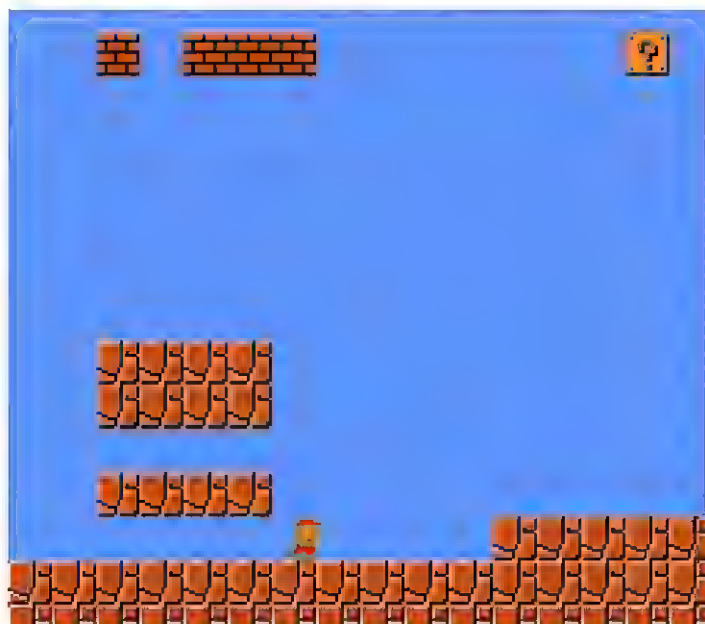


Figura 6.25: Imagen del primer prototipo

Gracias a la reescritura y realización de este nuevo prototipo he aumentado mi conocimiento, comparado con el que tenía al principio de todo, sobre el funcionamiento y comportamiento de las rutinas básicas que ejecuta la NES en cada frame. Ciertamente una de las cosas que más me ha costado es acostumbrarme a la utilización del lenguaje ensamblador 6502, aunque no por ser un lenguaje de muy bajo nivel, sino porque anteriormente ya había programado un videojuego en lenguaje ensamblador Z80 para la CPC Retro Dev: The Crypt².

Dejando a parte la diferencia que hay en cuanto a número de instrucciones, la más visible es el número de registros disponibles en la CPU que viene a ser menos de la mitad de los que hay disponibles en el Amstrad CPC: A, X e Y. De todas formas, gracias a los pocos ciclos de acceso que hay sobre la zero-page, se podría considerar que la NES tiene 256 "registros más" a disposición.

El sistema de colisiones del personaje sobre el mapa ha sido la primera rutina "impor-

¹<https://web.archive.org/web/20190326192621/http://nintendoage.com/forum/messageview.cfm?catid=22&threadid=7155>

²<https://www.cpc-power.com/index.php?page=detail&num=15656>

tante” que he llegado a implementar, cosa que no descarto que en un futuro tenga que cambiar su comportamiento ya que mi conocimiento será mucho mayor al que tengo ahora. Con este sistema quería que fuera una rutina no muy costosa al poder ejecutarse en casi todos los frames de ejecución del juego.

Por otra parte no necesitaba utilizar un byte completo para cada sprite de la pantalla por el hecho de que solo necesito saber si hay un 0 o un 1, es decir un bit. Finalmente, como se observa en la figura 6.13, de la sección 6.7 del desarrollo: cada sprite en X tiene un bit relacionado, en cambio cada dos sprites en Y están ligados a un mismo bit. De esta forma tengo una detección de colisiones más precisa en el eje X por si en un futuro quisiera utilizar esta ventaja para algo.

En cuanto a los problemas encontrados, entiendo que son inconvenientes que se puede encontrar una persona novata al tratar en este tema, los más notables pueden ser los siguientes:

- Entender como la NES trata la ejecución de cada frame: 60 veces por segundo en la versión NTSC, y 50 en la versión PAL, es ejecutada una interrupción: Ésta es la NMI (Non Maskable Interrupt) (Anexo B.10 para más información). Básicamente todas las veces que es activada esta interrupción realiza un “salto” en la memoria a la rutina que hará de bucle principal del juego.

Si no se utiliza este sistema de interrupción el tiempo que puede haber entre frame y frame puede ser variable y llegar a volverse locos tanto la CPU como la PPU.

- A pesar de utilizar la guía anteriormente nombrada para empezar a programar en este sistema, tenía algunos errores de programación que daban lugar a una mala ejecución del videojuego: Muchos de los fragmentos de código que aparecían como ejemplo estaban mal escritos.

Como ejemplo, uno de los problemas que me volvió loco era la aparición de un sprite que no utilizaba para nada en la esquina superior izquierda, por alguna razón. Finalmente, y tras muchas vueltas, me di cuenta que era culpa de la rutina que limpiaba la parte de la RAM dedicaba a los sprites ya que establecía todos sus valores a 0, cosa que hacía que los 60 sprites restantes que no utilizaba tuvieran el mismo índice, el 0, y estuvieran en la posición (0,0) de la pantalla. La solución era solo cambiar el byte con el que se limpiaba esta zona.

6.13.2. Iteración 2 - Movimiento alrededor de la pantalla y el nivel

- **Fecha de inicio:** 18 de noviembre de 2019
- **Fecha de finalización:** 15 de diciembre de 2019
- **Duración:** 28 días
- **Descripción:**

En mi mente, aunque este sea mi primer videojuego realizado para la NES, quiero realizar un producto parecido a los videojuegos clásicos de plataformas: Quería que

quien jugase a mi videojuego tuviera una sensación de libertad alrededor de un nivel completo y no sobre una pantalla estática.

Por ello, avanzando el prototipo ya creado, el siguiente paso más lógico era implementar un sistema de scroll tal y como se explica en la sección 6.6. Cabe destacar que no está aún funcionando junto al sistema de colisiones, ya que debo realizar algunos cambios en el almacenamiento de los mapas de colisión tal y como se ha hecho con el almacenamiento de los datos del mapa.

El concepto de scroll dentro de la NES es algo que me costó entenderlo ya que en un principio no entendía como podría moverse la pantalla y dibujar columnas nuevas fuera del trozo de pantalla que estoy viendo: Aumentando el valor que se escribe en el registro 0x2005, que actúa como offset, puede "moverse" entre las distintas name tables como se muestra en la figura 6.26. Claro está, si se llega de nuevo a 0, hay que cambiar la nametable que se consigue como "principal".

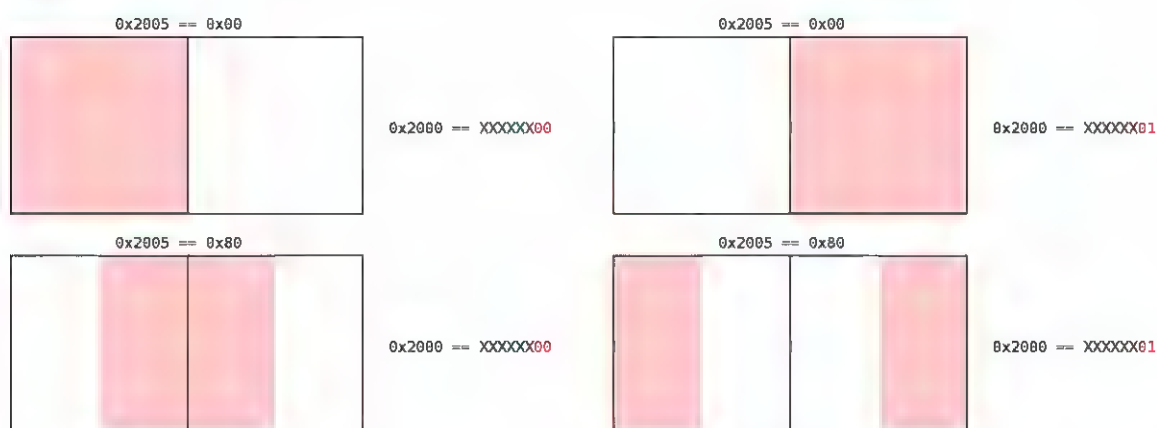


Figura 6.26: Comportamiento del valor que se introduce en 0x2005

He modificado el sistema de detección de la pulsación de los controles para que, fuera mucho más comprimido, óptimo y algo más flexible de lo que había anteriormente. En esta nueva rutina lo que se hace es lo siguiente:

1. Reiniciar el estado del registro del control del jugador 1.
2. Inicializar una variable a 1 que se va a utilizar como **ring counter**: En cada iteración del bucle se realiza una rotación de sus bits hacia la izquierda y se controla cuando el flag carry está activo. En el momento que esté activo, puede salir del bucle ya que se han hecho ocho iteraciones, es decir, se han comprobado todas las posibles pulsaciones. En la figura 6.27 puede verse un esquema de su funcionamiento.
3. En el propio bucle se realizan dos comprobaciones: Una es comprobar la pulsación del botón que toque en cada iteración, como se hacía antes, y otra es incrementar en dos una variable que se usa cuando se ha pulsado el botón en concreto.

Gracias al valor de esa variable, se obtendrá uno u otro valor según unos datos ya establecidos y almacenados de forma continua en memoria, como se ve en la tabla

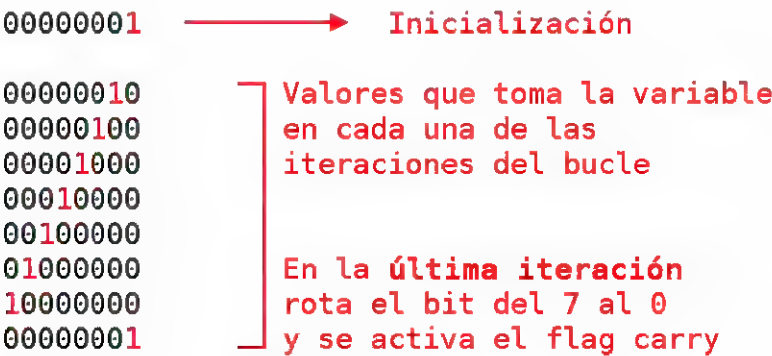


Figura 6.27: Esquema de funcionamiento del ring-counter utilizado para la lectura de los controles

6.6.

VAR	Byte devuelto	VAR	Byte devuelto
0x00	PRESS_A_LOW	0x01	PRESS_A_HIGH
0x02	PRESS_B_LOW	0x03	PRESS_B_HIGH
0x04	PRESS_SELECT_LOW	0x05	PRESS_SELECT_HIGH
0x06	PRESS_START_LOW	0x07	PRESS_START_HIGH
0x08	PRESS_UP_LOW	0x09	PRESS_UP_HIGH
0x0A	PRESS_DOWN_LOW	0x0B	PRESS_DOWN_HIGH
0x0C	PRESS_LEFT_LOW	0x0D	PRESS_LEFT_HIGH
0x0E	PRESS_RIGHT_LOW	0x0F	PRESS_RIGHT_HIGH

Tabla 6.6: Direcciones de memoria que se obtienen dependiendo del valor de la variable

A medida que pasaba la iteración, no es que me quedase sin memoria disponible, pero quería disponer de toda la memoria posible dentro de la disponible en la NES. Por ello, algo que yo había pasado por alto, pero que es algo muy importante, son los bancos de memoria: Fragmentos fijos de 8Kb en los que la memoria es dividida. En mi caso quería tener toda la memoria disponible con el mapper número 0, 32Kb, por lo que debo transmitirlo al compilador mediante el uso de la macro `.inesprg 2`.

Estos fragmentos de memoria son los que se intercambian en caso que se quiera realizar un *bank swapping* o intercambio de bancos, por ello debo tener en cuenta que no sobrepase el tamaño máximo del banco a la mitad de datos almacenados o a la mitad de una rutina, ya que puede causar comportamiento no definido.

Con el uso de 32Kb, debía crear cuatro divisiones visibles en el código tal y como se muestra en la figura 6.28. En el caso que alguna de esas secciones sobrepase los 8Kb, automáticamente lo que sobra es propiedad del siguiente banco de memoria.

Se puede ver también en la figura 6.28 que hay definido un banco extra para la PPU. Esto es debido a que el mapper número 0 tiene CHR RAM en vez de CHR ROM, eso quiere decir que los datos de los sprites se pueden incluir directamente sobre la PPU,

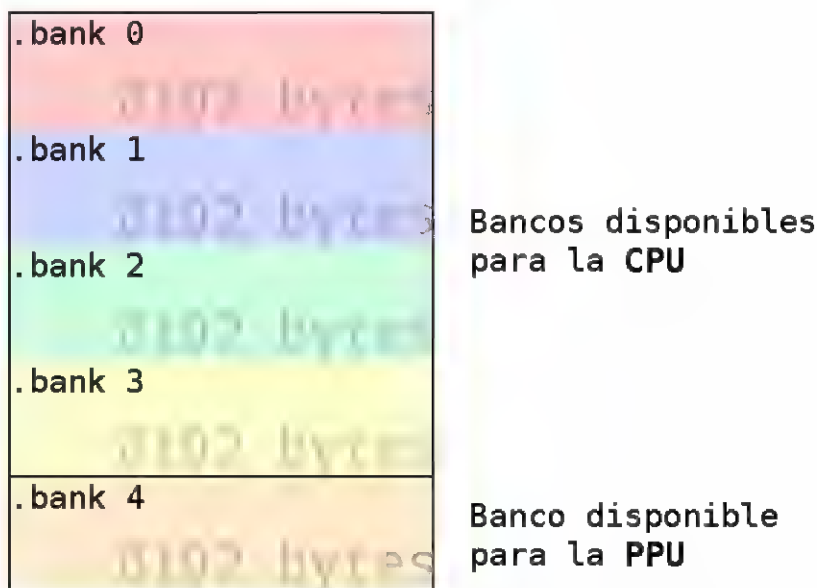


Figura 6.28: Esquema de división de la memoria en distintos bancos

por lo tanto el quinto banco que aparece pertenece a ésta. En el otro caso habría que hacer una rutina de transferencia de dichos datos de la CPU a la PPU.

6.13.3. Iteración 3 - Primer prototipo jugable

- **Fecha de inicio:** 16 de diciembre de 2019
- **Fecha de finalización:** 26 de enero de 2020
- **Duración:** 42 días
- **Descripción:**

El objetivo principal de esta iteración era tener un prototipo totalmente jugable en el que se pudiera mover el jugador, saltar, colisionar con el mapa y, como no, morir. Concretamente quedaban dos de esas tareas por implementar y otra, aunque obligatoria, ahora mismo opcional: Los enemigos.

Primeramente tenía la tarea de combinar el sistema de scroll del nivel con el sistema de colisiones, donde con éste último tuve que realizar varios cambios a la hora del almacenamiento y comportamiento de la rutina principal de comprobación.

Estos cambios residían en la forma que se almacena el mapa de colisiones: Puesto que el scroll se realiza en el eje horizontal, debía cambiar la forma en la que se guarda en memoria los datos de las colisiones ya que se tenían que leer en cuanto a columnas y no en cuanto a filas, como se muestra en la figura 6.29, al igual que realicé con el almacenamiento del mapa cuando se implementó el sistema de scroll.

En esa misma figura está marcado en rojo la ristra de datos con los que comprobaría la colisión en el caso que estuviera en la columna número cero. Si ese mismo fragmento

de datos lo conseguimos del sistema antiguo, sería totalmente un error, puesto que se miraría sobre los datos de una fila, y no una columna, y no sería posible iterar entre las columnas.

ANTIGUO	NUEVO	COLUMNA
00000000 00000000 00000000 00000000	00000000 00000000 00000000 00111111	0
00000000 00000000 00000000 00000000	00000000 00000000 00000000 00111111	1
00000000 00000000 00000000 00000000	00000000 00000000 11110011 00111111	2
00000000 00000000 00000000 00000000	00000000 00000000 11110011 00111111	3
00000000 00000000 00000000 00000000	00000000 00000000 11110011 00111111	4
00000000 00000000 00000000 00000000	00000000 00000000 11110011 00111111	5
00000000 00000000 00000000 00000000	00000000 00000000 00000000 00111111	6
00000000 00000000 00000000 00000000	00000000 00000000 00000000 00111111	7
00001111 11110000 00000000 00000000	00000000 00000000 00000000 00111111	8
00001111 11110000 00000000 00000000	00000000 00000000 00000000 00111111	9
00000000 00000000 00000000 00000000	00000000 00000000 00000000 00111111	10
00001111 11110000 00000000 00000000	00000000 00000000 00000000 11111111	11
00000000 00000000 00000011 11111111	00000000 00000000 00000000 00111111	12
11111111 11111111 11111111 11111111	00000000 00000000 00000000 11111111	13
11111111 11111111 11111111 11111111	00000000 00000000 00000000 11111111	14
11111111 11111111 11111111 11111111	00000000 00000000 00000000 11111111	15

Figura 6.29: Forma nueva de almacenamiento del mapa de colisiones

Una vez combinados los dos sistemas, me encontré con otra, aunque pequeña, piedra en el camino: En el momento que se dibujaba una nueva columna se añadía un offset para que apuntara a la siguiente columna de datos del mapa de colisiones. El problema venía porque solo aumentaba ese offset en el momento que dibujaba y entre medias, no hacía nada, por lo que resultaba que las colisiones se "desencajaban" de la pantalla.

La solución fue la suma o la resta de la posición del jugador, dependiendo de dos situaciones³:

Situación 1 Si el contador del scroll se encuentra entre 0 y 7 sumo los píxeles que faltan, puesto que **aún no se habría incrementado el número de columnas dibujadas**.

Situación 2 Si el contador del scroll se encuentra entre 8 y 15, entonces hago la resta de los píxeles, ya que **sí se habría incrementado el número de columnas dibujadas**.

Finalmente la muerte por caída del personaje, ya que por ahora no hay enemigos, no fue algo difícil de implementar, debido a que es una simple comparación, pero sí que se moría en más lugares de los que tenía permitido hacerlo. El problema venía dado al utilizar la instrucción `CMP #$F0` siendo un arma de doble filo. Para explicar como lo solucioné hay que mirar la tabla 6.7 con los resultados estableciendo cuatro alturas de ejemplo: `0x60`, `0x7F`, `0xC0` y `0xF3`.

- En principio se quiere que cuando sea mayor que el valor de corte se active la muerte, por lo que se puede pensar que salga de la propia rutina si el flag N no se activa. **Craso error**, este flag tampoco estará activo cuando el resultado de un valor menor que `0x80`.
- Finalmente la solución fue utilizar el flag C, puesto que si es mayor siempre se va a activar en el caso que se requiere ahora mismo.

³En la sección 6.7.2 se explican más detalles de como identificar estas situaciones

Altura	Flag Negative	Flag Carry
0x60	0	0
0x7F	1	0
0xC0	1	0
0xF3	0	1

Tabla 6.7: Explicación de la muerte del personaje por altura

6.13.4. Iteración 4 - Optimizando las rutinas y sistema de estados

- **Fecha de inicio:** 27 de enero de 2020
- **Fecha de finalización:** 23 de febrero de 2020
- **Duración:** 28 días
- **Descripción:**

En cuanto a trabajo global, esta ha sido una de las iteraciones más ajetreadas hasta ahora más que nada por el tiempo invertido en la solución de algunos problemas de rendimiento que aparecieron al final de la iteración anterior. Debido a la ejecución de muchas rutinas a la vez, se excedía el tiempo máximo de refresco de la pantalla que se dedica para cada frame. Esto ocasionaba que el scroll se ejecutara de forma incorrecta, puesto que lo hacía "fuera de su tiempo".

La solución final ha sido la utilización de las llamadas mediante código automodificable gracias a la utilización del modo de direccionamiento indirecto de la memoria y el uso de la instrucción JMP.

Gracias a estas herramientas, en el momento que no es utilizada una rutina, porque no tiene porqué, dicha dirección de memoria no apunta a la rutina, sino a otra que solo contiene la instrucción RTS para que en cuanto entre, salga directamente y no pierda ciclos tontamente, tal y como se muestra en la figura 6.30.

Cabe decir que dichos problemas aparecían cuando se emulaba el juego en la versión NTSC al funcionar a un refresco de 60Hz, mientras que en la versión PAL no, por funcionar a 10Hz menos y el tiempo máximo de refresco de la pantalla en cada frame es un poco mayor. De todas formas quiero que este proyecto funcione en ambas versiones de forma correcta, por lo que siempre busco que vaya perfecto en el sistema con más limitaciones.

Seguidamente, quería que cada estado del juego tuviera una rutina de actualización distinta según las necesidades de cada uno de ellos. Por ejemplo, en la pantalla de inicio no se van a estar ejecutando las rutinas de comprobación de colisiones ni de scroll, por poner algunas, sino que solo va a estar esperando la pulsación de un botón para empezar el juego.

Por ello, utilizando el mismo sistema que el punto anterior, he implementado un sistema de estados que, con la modificación de solo los dos bytes, correspondientes a la dirección



Figura 6.30: Desactivación de la llamada a una rutina

de memoria de la rutina destino, permite cambiar entre distintos procesos de actualización del juego en cualquier momento. El esquema de funcionamiento se encuentra en la figura 6.31.

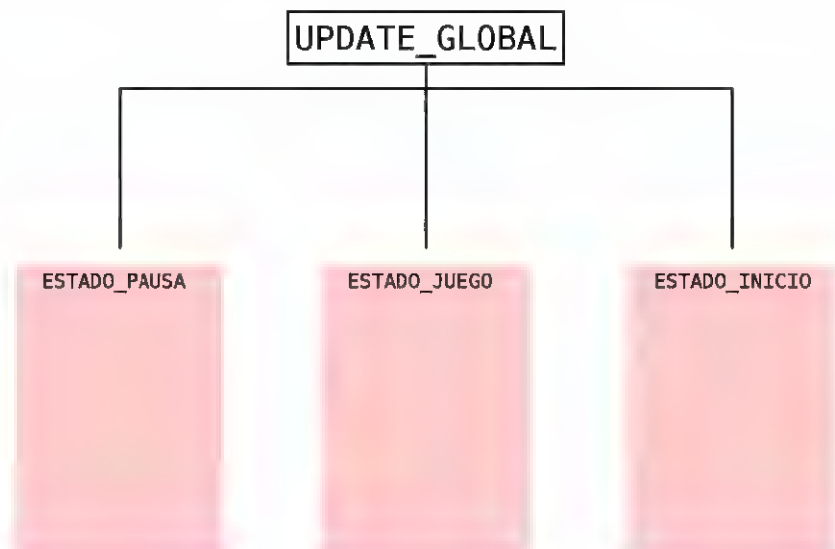


Figura 6.31: Esquema de funcionamiento del sistema de estados

Gracias a este sistema puedo tener un comportamiento totalmente distinto y disponer de los medios que cualquiera de los estados requiera sin afectar al rendimiento general del juego.

Puesto que el juego ya se encontraba en una situación donde las mecánicas del personaje principal y del propio escenario estaban ya implementadas, era hora de que aparecieran enemigos por el todo el nivel, por lo que fue lo siguiente que hice. Con estos tenía dudas

en cuanto al rendimiento ya que habrán varios enemigos siempre en pantalla. Además, no puedo poner más de tres enemigos a la misma altura por el problema del límite de los ocho sprites máximos por scanline, por lo que finalmente solo podrían aparecer un total de tres enemigos a la vez en pantalla.

La creación de enemigos fue una tarea sencilla:

1. Como sabía las direcciones de memoria donde finalmente irían los sprites de los enemigos, compruebo si el primer byte de una de las tres filas de 16 bytes es 0xFE, ya que se que ese valor será el de "no hay enemigo, puedes crearlo aquí". En el caso que no encuentre en ninguna de las tres filas, porque ya hay tres enemigos, no se crea ninguno más tal y como se observa en la figura 6.32.

```
000200: C0 01 00 78 C0 09 00 80 C8 11 00 78 C8 19 00 80 > Personaje principal
000210: FE FE FE FE FE FE FE FE FE FE FE FE FE FE FE FE \ Memoria reservada
000220: FE FE FE FE FE FE FE FE FE FE FE FE FE FE FE FE > para la creación
000230: FE FE FE FE FE FE FE FE FE FE FE FE FE FE FE FE / de nuevos enemigos
```

Se crea un enemigo nuevo

```
000200: C0 01 00 78 C0 09 00 80 C8 11 00 78 C8 19 00 80
000210: 60 28 01 96 60 20 01 9E 68 38 01 96 68 30 01 9E
000220: FE FE FE FE FE FE FE FE FE FE FE FE FE FE FE FE
000230: FE FE FE FE FE FE FE FE FE FE FE FE FE FE FE FE
```

```
000200: C0 01 00 78 C0 09 00 80 C8 11 00 78 C8 19 00 80
000210: 60 28 01 96 60 20 01 9E 68 38 01 96 68 30 01 9E
000220: 60 28 01 96 60 20 01 9E 68 38 01 96 68 30 01 9E
000230: 60 28 01 96 60 20 01 9E 68 38 01 96 68 30 01 9E
```

No se pueden crear más enemigos

Figura 6.32: Estado de la memoria en el momento de la creación de un enemigo

2. En el momento que es creado el enemigo, se establece la dirección de memoria de la rutina de actualización que dicho enemigo ejecutará.

También se establecen en memoria los datos de los sprites que utiliza el enemigo de unos de ejemplo como se ve en el fragmento de código 6.12. Se puede observar que no solo están los índices de los sprites que el enemigo tiene que utilizar, sino que también se encuentran, en la primera y última columna, los offset que hay que añadir a la altura en la que se crea dicho enemigo.

Estos offsets son para que, sencillamente, cada sprite aparezca en la posición correcta, por lo que si hay que establecer el segundo sprite en memoria, hay que sumarle 8 para que salga a la derecha del primero.

Código 6.12: Datos del tipo de enemigo número uno

```
T1_ENEMY:
    .db $00, $28, $01, $00
    .db $00, $20, $01, $08
    .db $08, $38, $01, $00
    .db $08, $30, $01, $08
```


Tal y como se indica en el punto uno, se establece una rutina de update para los enemigos. El código está preparado para poder cambiar la dirección de memoria de la rutina a donde apunta. De todas formas, hay una operación obligatoria que se debe aplicar a todos los enemigos: Añadir un offset, que aumenta a medida que se hace scroll, a la posición para que de la sensación que cada enemigo se mueve por el nivel y no por la pantalla.

Como se muestra en la figura 6.33 si no se añade dicho offset, el enemigo siempre va a estar en la misma posición de la pantalla y no dará la sensación de que su entorno es el nivel.

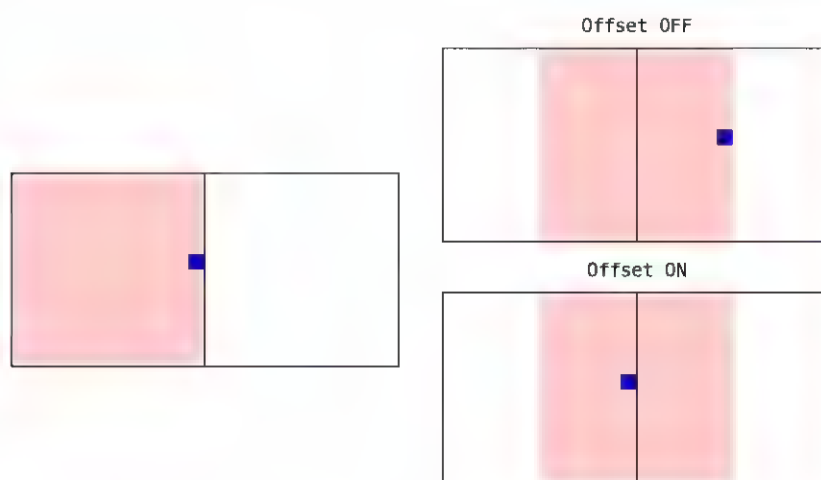


Figura 6.33: Esquema de adición del offset a los enemigos para un movimiento alrededor del nivel

Cabe decir, que en el caso de los enemigos, puesto que puede darse la posibilidad que hayan varios a la vez en pantalla, todas las acciones de su update están limitadas para que se ejecuten en el momento que toca y no en todos los frames. Por ejemplo con las colisiones: Puesto que las colisiones son fijas, cada 16 píxeles es comprobado si se choca o no.

Finalmente, para cambiar el movimiento del que las entidades disponen ahora, han sido cambiadas todas las rutinas de movimiento y de salto de los enemigos y el personaje principal por el uso de tablas de saltos: Dependiendo del valor de una variable que actúa como contador, se consigue que en cada frame el personaje se desplace un número distinto de píxeles que ya están preestablecidos en memoria. Como ejemplo, en la tabla 6.8 se muestran unos valores de como sería una rutina que muestra este comportamiento:

```
JUMP_TABLE: .db $01, $02, $00, $03, ...
```

Al terminar con la implementación, apareció un pequeño problema a la hora de aplicar el scroll con el nuevo sistema de movimiento: Tocaba cambiar la forma en la que se comprobaba cuando había que dibujar una nueva columna en el scroll y cuando no.

En el antiguo sistema de movimiento siempre aumentaba la X y la Y del personaje en 1, era un movimiento totalmente lineal y por lo tanto no había problemas a la hora

Contador	Valor conseguido
0	0x01
1	0x02
2	0x00
3	0x03
...	...

Tabla 6.8: Ejemplo de funcionamiento de una tabla de saltos

de comprobar si había que dibujar una columna nueva. Pero claro, en el momento que se cambia de sistema de movimiento, como muestra el frame 6 de las tablas 6.9, podía dar la casualidad que dicho nuevo valor nunca fuese 8 y por lo tanto nunca entrar a la rutina que dibujaba las nuevas columnas.

Contador	Valor	Número de frame	Valor acumulado
0	0x01	1	0x01
1	0x02	2	0x03
2	0x01	3	0x04
3	0x02	4	0x06
4	0x01	5	0x07
5	0x02	6	0x09

Valores a obtener de la tabla de saltos Acumulación del valor sobre la variable auxiliar

Tabla 6.9: Tablas que muestran el error en el funcionamiento del scroll con el nuevo sistema de movimiento

Por lo tanto en la rutina de movimiento del personaje la variable auxiliar de scroll **aumenta de 1 en 1, tantas veces como indique el valor obtenido por la tabla de saltos.**

6.13.5. Iteración 5 - Cambiando el estilo visual

- **Fecha de inicio:** 24 de febrero de 2020
- **Fecha de finalización:** 23 de marzo de 2020
- **Duración:** 28 días
- **Descripción:**

A mitad de esta iteración conseguí, después de unos meses detrás de ella, una NES real para probar el videojuego, ya que podría cambiar mucho de ejecutarlo en un emulador al sistema real para el que estaría hecho: Así fue, la primera ejecución fue un desastre. En iteraciones anteriores había cambiado el orden de algunas rutinas del update del videojuego, cosa que fue por la que empecé a buscar y solucionar dicho problema.

Poco a poco y a base de prueba y error, puesto que solo encontré un par de publicaciones en foros sobre temas parecidos, descubrí que dentro del propio bucle de ejecución del juego las propias rutinas deben tener un orden específico. Hasta ahora las rutinas que cambiaban datos de la PPU estaban "mezcladas" con las rutinas que actualizaban la lógica del juego.

La solución fue cambiar dicho orden en el bucle principal como se muestra en la figura 6.34: Primero de todo se ejecutan las rutinas que cambian datos de la PPU, después se establecen los distintos valores sobre los registros que ésta utiliza y finalmente se ejecutan las rutinas que tratan la lógica del juego.

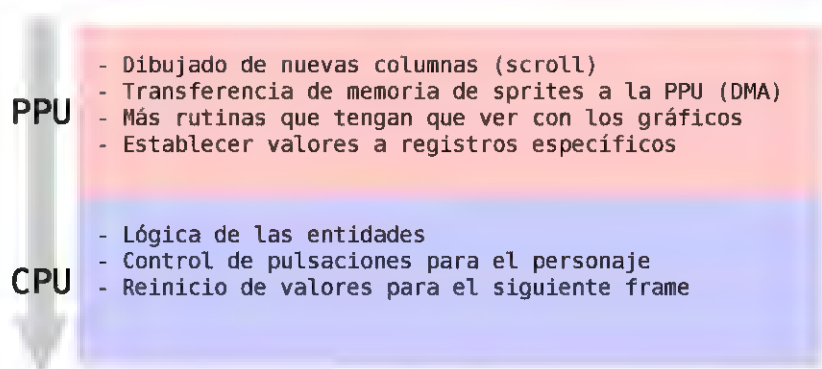


Figura 6.34: Orden correcto de ejecución de las rutinas en el bucle principal

Seguidamente, utilizando el mismo sistema que realicé en la iteración anterior para la creación y actualización de los enemigos, en esta he utilizado el mismo principio para los objetos que el usuario tendrá que recoger alrededor de los niveles. La única diferencia es la lógica de ambas entidades ya que los objetos no van a poder moverse por ellos mismos.

Una vez implementados los objetos, terminé el diseño de todos los gráficos: Tanto los sprites definitivos que compondrán el escenario como los de las distintas entidades, tal y como se muestran en las figuras 6.35. Además, aunque aún no estuvieran implementadas, he diseñado las dos animaciones principales del personaje que controla el usuario.



Figura 6.35: Sprites definitivos

La apariencia del personaje principal fue lo primero que me puse a diseñar ya que sabría que a partir de ahí podría definir un poco mejor todo lo demás. Mi idea es que no fuera algo "animal", por ello lo primero que me vino a la mente fue hacer una especie de

”gota”, recordando un poco el diseño de Kirby, como se muestra en la figura 6.36.

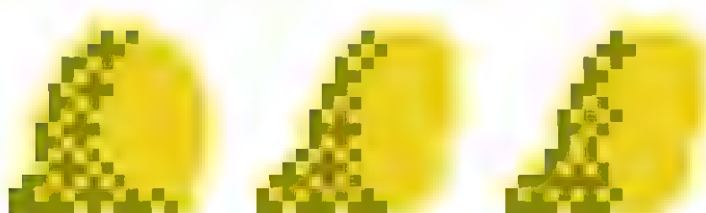


Figura 6.36: Diseño inicial del personaje principal

Al terminarlo, lo seguí utilizando como diseño provisional para el juego aunque fuera algo que no me convencía y quería cambiarlo totalmente. Finalmente, como se ve en las figuras 6.35a y 6.35b, opté por una recoloración mínima del sprite completo y por usar una forma más o menos humana.

A partir del momento que terminé los diseños principales del personaje y los enemigos, cambié por completo la estética general de los niveles, como se ve en la figura 6.37: De un estilo muy oscuro a otro un poco más lúcido y con el uso de varias paletas de colores dentro de las que había establecido.

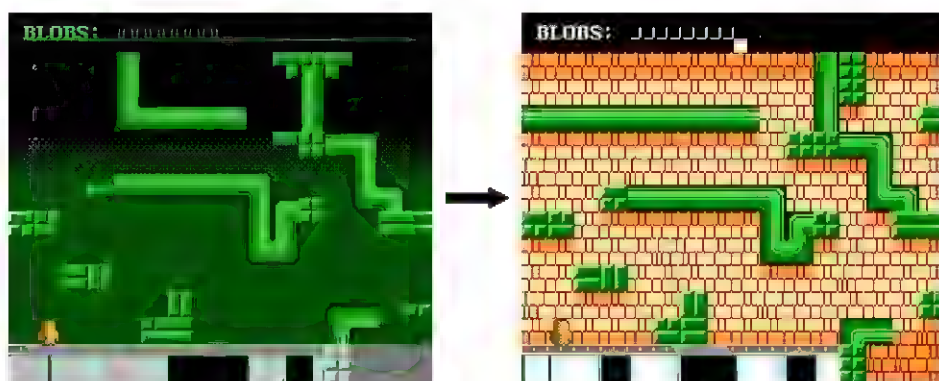


Figura 6.37: Cambio del estilo general de los niveles

Se puede observar en la imagen de la derecha de la figura 6.37 que en la parte de arriba se muestra el número de ítems que el jugador ha recogido en todo el nivel. Para ello ha sido falta el uso del flag de colisión que hay con el sprite número 0, es decir, el que ocupa las direcciones de memoria desde 0x0200 hasta 0x0203.

He decidido el uso de esta característica por profundizar más en las características que puede ofrecer la NES y otra por que permite tener una zona de la pantalla que no va a ser actualizada nunca, a no ser que yo mismo cambie los datos directamente, además que para dibujar esta zona se utilizan la pattern table para el fondo, por lo que hay que olvidarse del límite de los 8 sprites por scanline.

En sí no ha sido una tarea complicada de implementar, aunque sí que ha habido un

pequeño problema al aprender sobre su funcionamiento ya que, como aviso en la sección 6.9 del desarrollo, en principio se quedaba en un bucle infinito debido a que el sprite que usaba el color definido como transparente.

Ya para terminar con el diseño en general del juego, diseñé un set entero de sprites para la pantalla de inicio del juego: Quería que ésta aprovechara al máximo los 256 sprites que están disponibles. Finalmente utiliza un total de 227 como se ve en la figura 6.38. En el caso que no cupieran todos, tenía pensado utilizar sprites sueltos por la pantalla, respetando los 8 sprites por scanline, colocados de forma estratégica.



Figura 6.38: Diseño de la pantalla inicial

En la parte final de la iteración, centré mi trabajo en optimizar tanto el rendimiento del videojuego como en reducir el tamaño que los datos de cada nivel ocupaban en memoria:

- El videojuego ya tiene algunas rutinas costosas, como el dibujado de nuevas columnas o la propia creación de los enemigos. He decidido partir la ejecución de muchas de esas rutinas para que se ejecuten 3 o menos frames más tarde de cuando deberían, cosa imperceptible para la persona que está jugando al videojuego, pero muy saludable para respetar los ciclos máximos que establece la NES en cada frame. De esta forma se divide el trabajo que se haría en un frame para hacerlo en varios.
- Cada nivel puede llegar a ocupar un total de 8192 bytes en memoria, sin contar mapa de colisiones ni los datos de los colores para las attribute tables. Eso quiere decir que para cada nivel, haría falta un banco entero, cosa que es totalmente inviable.

Debido a esto he decidido realizar una compresión del mapa en cuanto a metasprite: Guardar la información de 4 sprites en 1 byte, en definitiva reducir el tamaño total del nivel a un cuarto de su original. Podría haber escogido algún algoritmo como

RLE, pero me he decantado por hacer este sistema de metasprites por varias razones:

- * No aplico ningún algoritmo de compresión de datos, por lo que en un futuro se podría implementar uno sin problemas.
- * Puesto que el almacenamiento de los mapas se realiza en sentido vertical, la repetición de bytes es algo menos improbable de como podría ser con un sentido horizontal, pudiendo la información llegar a ocupar más, al aplicar el algoritmo.

He tenido que cambiar varias rutinas del juego y de un exportador de niveles comenzado unas iteraciones atrás: Realizado en C++, a partir de un `.csv` exportado desde Tiled, genera tres archivos distintos con la **información**, los **atributos** y las **colisiones** de un nivel entero.

De entre las rutinas que han cambiado su comportamiento, la más reseñable ha sido aquella que se encarga de pintar nuevas columnas, puesto que le afectan directamente los datos que son conseguidos. Los metasprites son almacenados en memoria en sentido vertical, como se ve en la figura 6.39. De esta forma se en todo momento que sprites y de que columna, izquierda o derecha, hay que conseguir y escribir en la PPU.

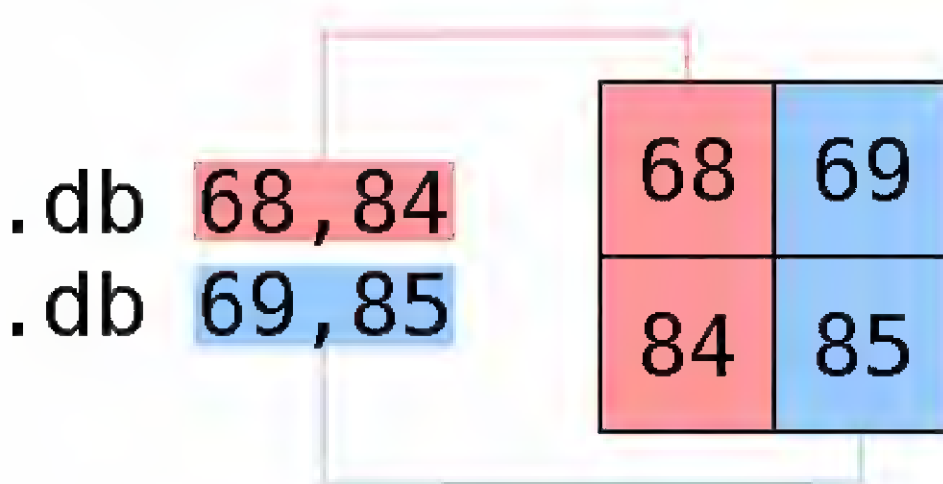


Figura 6.39: Organización de los datos de cada metasprite

Por otra parte, para utilizar el exportador hay que seguir los siguientes pasos:

1. Definir cuales son los metasprites que se van a usar, sabiendo el índice de los sprites que conforman cada uno de ellos, tanto en código como en Tiled (Figura 6.40).
2. Definir qué paleta utiliza cada uno de los sprites y cuáles tendrán o no colisión (Figura 6.41).
3. Pre-crear el mapa en Tiled y exportarlo en `.csv` (Figura 6.42).

Al pasarle el archivo al programa leerá cada uno de los valores convirtiéndolos a hexadecimal y dejando el archivo listo para arrastrarlo a la carpeta que se

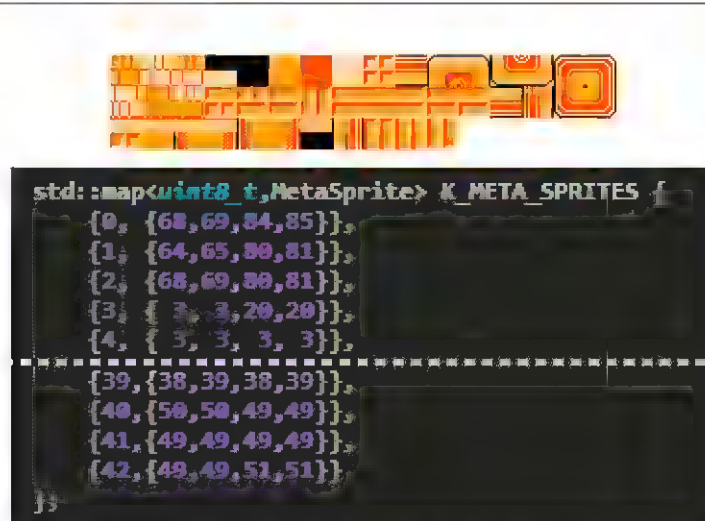


Figura 6.40: Metasprites de los niveles definidos para su uso

deseo.

Realmente una vez se han hecho los dos primeros pasos, que son los más tediosos por el hecho de tener que definir la información que se va a utilizar, el cambio de cualquier cosa aspecto del nivel es casi instantáneo y útil para un mejor y más rápido desarrollo del videojuego.

6.13.6. Iteración 6 - Mejora del diseño del juego

- **Fecha de inicio:** 23 de marzo de 2020
- **Fecha de finalización:** 12 de abril de 2020
- **Duración:** 21 días
- **Descripción:**

Primeramente, antes de empezar a hacer nada nuevo, centré mi trabajo en la construcción de la pantalla de inicio con el nuevo sistema de compresión y descompresión de metatiles que estaba implementado y que se explica en la sección 6.11: El trabajo realizado ha sido un poco más tedioso de lo normal puesto que la mayoría de las metatiles compuestas son distintas, tal y como se muestra en la figura 6.43.

A la hora de la ejecución del juego con la nueva pantalla de inicio surgió un problema que afortunadamente, gracias al debugger del emulador que utilizo, no tardé más de media hora en arreglar: Por alguna razón el tiempo que se tardaba en pintar toda la pantalla de inicio y en establecer las variables necesarias para el inicio de todo el juego era mayor que el tiempo que la NES dedica a la vblank, por lo que habían variables que no llegaban a iniciarse, como la que establecía la rutina del update que ejecutaba al saltar la interrupción NMI y quedaba la ejecución en un bucle infinito.

La solución, a parte de cambiar de orden algunas rutinas, ha sido la de establecer el valor de algunas variables en la zona de la memoria donde no está la interrupción NMI

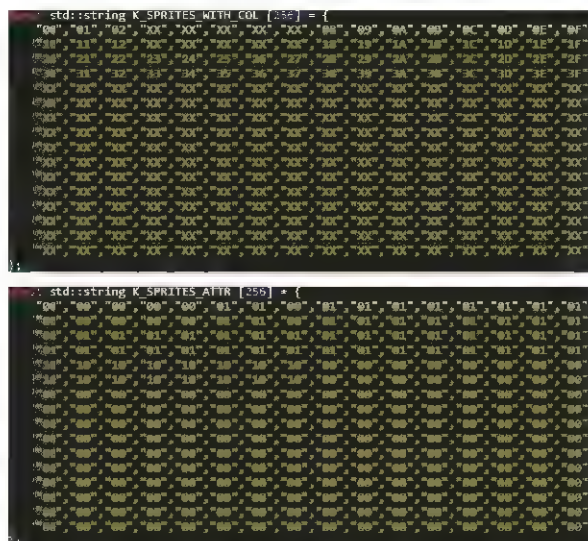


Figura 6.41: Datos de las colisiones y los atributos de cada uno de los sprites



Figura 6.42: Construcción del mapa en Tiled

activada y se pierde un número muy grande de ciclos.

El comportamiento del exportador ha sido modificado para poder elegir entre varios sets de metatiles para la construcción del archivo final. Estos sets de metatiles están separados según permite el mapper que tengo elegido, es decir cuatro sets en total, debido al mapper número tres, de los que, por ahora están utilizados dos de ellos, como se ve en el fragmento de código 6.13.

Código 6.13: Parámetros disponibles para el exportador

```
$ ./main.exe
Error en el envío de argumentos.
Debe ser: ./main.o archivo.csv num_pantallas banco_elegido prefijo
    - num_pantallas: 1 pantalla es 16 metasprites en horizontal (8 = 1 nivel entero)
    - banco_elegido:
    - 0 -> Banco de nivel
    - 1 -> Banco de pantalla de inicio
    - prefijo: L1 para el primer nivel, L2 para el segundo, ...

$ ./main.exe pantalla_inicio_con_metatiles.csv 8 2 prueba
Error en el banco elegido
```

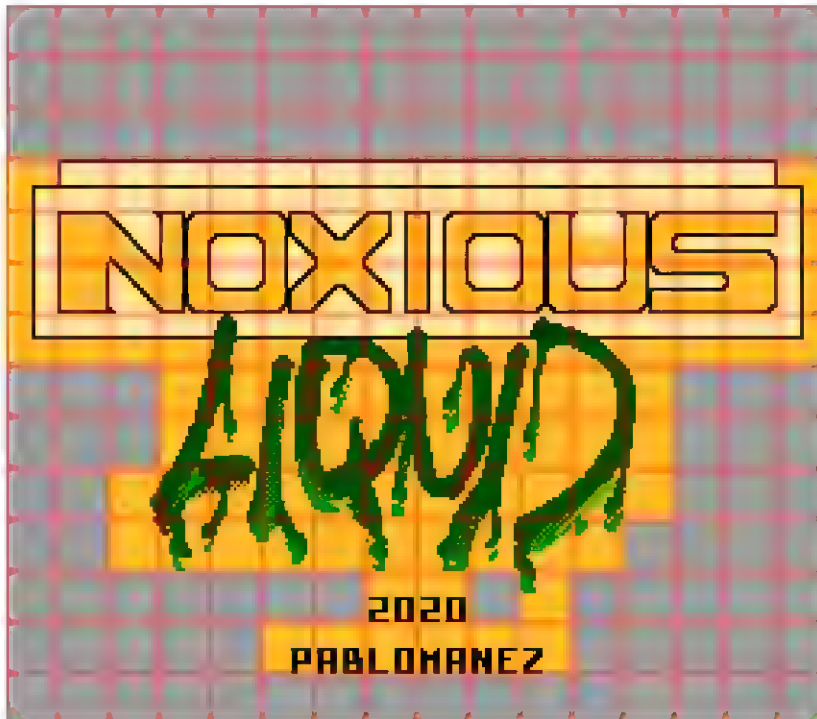



Figura 6.43: Composición de las metatiles en la pantalla de inicio

```
- Banco no definido
$ ./main.exe pantalla_inicio_con_metatiles.csv 8 3 prueba
Error en el banco elegido
- Banco no definido
```

He realizado algunos cambios menores en el código:

1. Se ha modificado la forma en la que los enemigos cambiaban la dirección de movimiento: Anteriormente solo cambiaban de dirección si encontraban un bloque a su derecha o izquierda que chocara contra ellos. Ahora mismo también cambian la dirección en el momento que encuentran un bloque vacío en la diagonal según la trayectoria que lleven en ese momento, según muestra la figura 6.44.

De esta forma se le da un poco más de dinamismo y variedad a los niveles puesto que antes era obligado colocar un bloque de colisión que chocara con su trayectoria para hacerlo girar.

2. El sistema de colisiones es una de las primeras rutinas "importantes" que implementé. En su momento no tenía la destreza que tengo ahora mismo para programar en lenguaje ensamblador 6502 y era de esperar que tuviera sus límites. Concretamente, ese límite era encontrado al recorrer justo la mitad del nivel: Cada nivel ocupa 8 pantallas y cada pantalla tiene asignada un mapa de colisiones de 64 bytes. Si se hacen algunas cuentas rápidas ya se puede observar que para recorrer todo el mapa de colisiones se necesitan 2 bytes, puesto que de principio a fin el mapa de colisiones ocupa 512 bytes, cosa que no se hacía al principio y solo se



Figura 6.44: Muestra del comportamiento nuevo de los enemigos

utilizaba un byte.

Para paliar este problema, a parte de darle un pequeño repaso de optimización a la propia rutina, he utilizado una serie de variables temporales que están definidas y que las puedo utilizar en cada rutina, ya que al final de ésta van a estar "libres".

Como se ve en la figura 6.45, el cambio reside en el momento de calcular la dirección a la que debo atacar, puesto que antes acumulaba todo al registro Y.

Método antiguo

```
REG_A      = (POS_X + NUM_COLUMNAS)*4
REG_A      += POS_Y
REG_A      --> REG_Y
```

} No hay control sobre el flag Carry

Método nuevo

```
TMP_ALTO  = COL_ALTO
TMP_BAJO  = (POS_X + NUM_COLUMNAS)*4
TMP_BAJO += COL_BAJO
```

} Si en cualquiera de estas dos operaciones se activara el flag carry:
TMP_ALTO++

(*) COL_ALTO y COL_BAJO son, respectivamente, los bytes alto y bajo que apuntan al inicio del mapa de colisiones del nivel

Figura 6.45: Esquema de control de la activación del flag carry en la rutina de colisiones

3. He implementado un sistema de cambio y almacenamiento de niveles en el que se establecen una serie de variables a unos valores fijos durante la ejecución de todo el nivel o estado, en el caso de la pantalla de inicio. Concretamente está organizado mediante el uso de los bytes bajos y altos de cada dirección de memoria a los datos en sí. De esta forma puedo apuntar a cada una de las zonas de datos que se necesiten en cada momento con el uso de una sola variable, que es la que indica el número de nivel en el que se encuentra el jugador en dicho momento.

He realizado también el primer nivel definitivo, como se muestra en la figura 6.46,

teniendo en cuenta un sistema de progresión moderado y que sea fácil para que el jugador aprenda a moverse por los niveles.



Figura 6.46: Nivel 1 definitivo

Finalmente, he acabado de diseñar las animaciones de los enemigos, como se muestra en la figura 6.47, y he implementado una rutina para que con establecer un valor en tres variables y el registro Y sea posible animar cualquier entidad compuesta por cuatro sprites seguidos en memoria.



Figura 6.47: Animaciones de los enemigos

En la figura 6.48 se explica el funcionamiento de dicha rutina. En un principio pensé en realizar un almacenamiento de los sprites seguido en las pattern table para que al final se cambiase de sprite con solo aumentar su valor, pero entonces no se me ocurría ninguna solución para cuando hubiera que girar los sprites que duplicar sprites tontamente, por lo que al final he utilizado el uso de tablas de animación para que fuera, además, más estándar a todas las entidades.

He decidido implementarlo de esa forma puesto que todas los enemigos y el héroe tienen el mismo número de sprites seguidos en memoria. En el caso que decidiera implementar alguna entidad con unas características distintas, habría que cambiar el comportamiento de la rutina. Puesto que además es una rutina que va a ejecutarse muy frecuentemente, quería que fuera lo más compacta posible para que tardase el mínimo número de ciclos posible.

Las zonas de memoria que aparecen en la parte derecha de la figura 6.48 son, respectivamente, los índices que corresponden a los sprites que, en este caso, componen la animación de movimiento del héroe y la zona de inicio de la RAM dedicada a los sprites. En este último están marcados los sprites que corresponden al héroe, por lo que en el caso que se quiera animar algún enemigo u otra entidad distinta que cumpla las mismas características, hay que cambiar esa dirección de memoria.

A la hora de hacer el mirror de los sprites, y que apunten a la dirección contraria, quería utilizar la misma rutina que he escrito para las animaciones hacia un lado, además que

ENTRADAS

TABLA_ANIM : Tabla de animaciones a usar
 DIR_SPRITE : Dirección de memoria a modificar
 TOTAL_ANIM : Número total de frames (*)
 REG_Y : Frame actual

FUNCIONAMIENTO

```

TMP_VAL1 = 0
do{
    REG_A      = TAB_A_ANIM+REG_Y
    TMP_VAL2   = REG_Y
    REG_Y      = TMP_VAL1
    DIR_SPRITE+REG_Y = REG_A

    TMP_VAL1   = REG_Y+4
    REG_Y      = TMP_VAL2+TOTAL_ANIM
}while(TMP_VAL1 < 16)
  
```

ANIM_HERO_MOVE:	Frame 1
ANIM_HERO_MOVE_TL:	.db \$02, \$00, \$03, \$00
ANIM_HERO_MOVE_TR:	.db \$0A, \$08, \$0B, \$08
ANIM_HERO_MOVE_BL:	.db \$12, \$10, \$13, \$18
ANIM_HERO_MOVE_BR:	.db \$1A, \$18, \$1B, \$18

(*) Número total de frames

Dirección de memoria de inicio

000200:	17 FF 01 80 C0 00 00 10 C0 08 00 18 C8 10 00 10
000210:	C8 18 00 18 FE FE FE FE FE FE FE FE FE FE FE

Figura 6.48: Esquema de ejecución de la rutina de animaciones

no quería otra rutina, repitiendo código y que fuera muy costosa. Por lo que al final, con la consecuencia de sacrificar algunos bytes, he acabado almacenando los datos de los mismos sprites pero en orden cambiado y seguidos a los que ya había, tal y como se ve en el fragmento de código 6.14. De esta forma, solo tengo que sumar un offset, fijo para toda la animación, y ya tendría el sprite bien construido.

Código 6.14: Almacenamiento de los datos de los sprites

```

ANIM_HERO_MOVE:
    ANIM_HERO_MOVE_TL: .db $02, $00, $03, $00
    ANIM_HERO_MOVE_TR: .db $0A, $08, $0B, $08
    ANIM_HERO_MOVE_BL: .db $12, $10, $13, $18
    ANIM_HERO_MOVE_BR: .db $1A, $18, $1B, $18
ANIM_HERO_MOVE_MIR:
    ANIM_HERO_MOVE_MIR_TL: .db $0A, $08, $0B, $08
    ANIM_HERO_MOVE_MIR_TR: .db $02, $00, $03, $00
    ANIM_HERO_MOVE_MIR_BL: .db $1A, $18, $1B, $18
    ANIM_HERO_MOVE_MIR_BR: .db $12, $10, $13, $18
  
```

Finalmente, para que el mirror este acabado y se muestre bien, he creado una rutina que lo único que hace es cambiar el byte de atributos según una dirección de memoria que se le pase como entrada: Dicho cambio es un simple OR Exclusivo con el bit 6, para invertir su valor y de los tres sprites siguientes.

6.13.7. Iteración 7 - Videojuego terminado

- **Fecha de inicio:** 13 de abril de 2020
- **Fecha de finalización:** 3 de mayo de 2020
- **Duración:** 21 días
- **Descripción:**

Debido al poco tiempo que me queda de desarrollo del juego, esta es la última iteración que voy a utilizar para introducir nuevo contenido al videojuego además de realizar los

últimos retoques grandes al videojuego, a falta de pequeños detalles que se queden para el último tramo de desarrollo.

Antes de seguir con el desarrollo del videojuego, cabe decir que la primera semana la dediqué a la creación de distintas secciones nuevas en el desarrollo y la mejora de las explicaciones, tanto en el desarrollo como en la propia sección del seguimiento del proyecto. Una vez hecho esto, para una mejor organización, dividí el resto del desarrollo en tres partes: Una en la que **arreglara algunos detalles que faltaban por pulir y añadir**, otra en la que entraba la **remodelación del comportamiento del movimiento de los enemigos** y por último la **creación de todos los niveles del juego**.

En esta primera parte realicé los siguientes cambios:

- **Desarrollo del juego:** Una cosa que me faltaba, y que resulta bastante obvia, fue indicar que en el momento que el personaje llegaba al final del nivel se cargara el siguiente nivel automáticamente, ya que antes se quedaba parado en el momento que llegaba a cierta distancia del inicio. Además, en el momento que falte por recoger al menos uno de los ocho orbes que hay en cada uno de los niveles, al llegar al final éste es reiniciado para tener otra oportunidad de recogerlos de nuevo.

También, como se observa en la figura 6.49, y para mejorar la realimentación del jugador, he añadido un indicador en forma de texto en la parte de arriba de la pantalla que muestra el nivel en el que se encuentra ahora mismo.



Figura 6.49: Indicador del nivel actual en la parte superior de la pantalla

Finalmente, he añadido en la pantalla de inicio un "PRESS START", como se muestra en la figura 6.50 para indicar al jugador cual es la tecla que debe pulsar para iniciar el juego, que puede ser muy obvio, pero hay que guiar el jugador en todo momento.

He decidido utilizar sprites para escribir este texto por el hecho de ser solo 5 sprites en fila, cosa que no le afecta el límite de sprites por scanline, y además porque he querido añadir un pequeño parpadeo al texto de la forma más sencilla posible: Cada X ciclos cambio paleta de colores que utilizan esos sprites de A a B y viceversa, como se muestra en la figura 6.51.

- **Arreglo de bugs:** He arreglado algunos fallos y situaciones que faltaban por comprobar como:
 - * Que el personaje chocara con la parte izquierda de la pantalla.
 - * Cuando se moría mirando hacia la izquierda, al reaparecer las animaciones del personaje se mostraban mal.

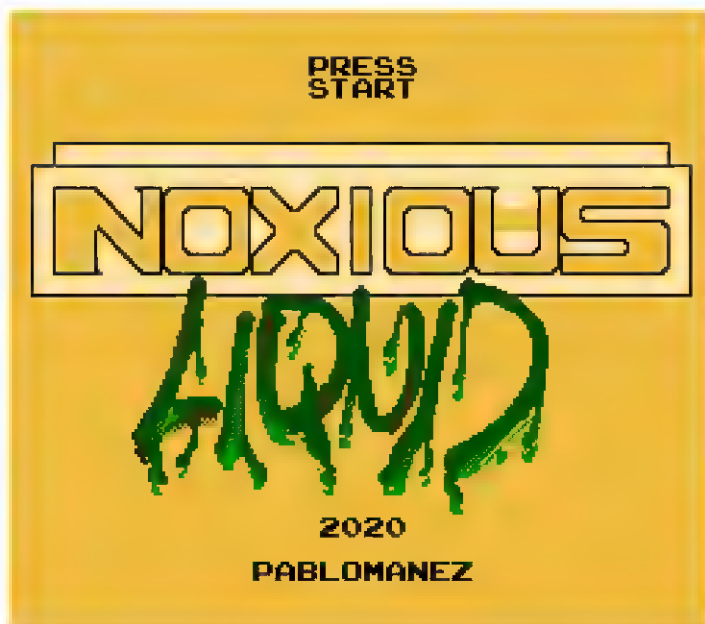


Figura 6.50: Texto "PRESS START" en la pantalla de inicio



Figura 6.51: Paletas utilizadas en el parpadeo de "PRESS START" en la pantalla de inicio

- * Cuando se pulsaba izquierda o derecha y justo después su opuesto cabía la posibilidad de seguir moviéndose hacia la primera dirección que se había pulsado.

En cuanto a la creación de los niveles, en principio tenía en mente crear todos los niveles posibles hasta llenar la memoria: En total tengo casi 3 bancos de memoria vacíos y cada uno de los niveles ocupa un total de 3072 bytes a falta de la aplicación de más compresión sobre el almacenamiento de los datos, por lo que podría haber hecho un total de 8 niveles. Al final he realizado un total de 6 niveles como se muestra en la figura 6.52.

A mitad del diseño de los niveles, se me ocurrió crear una especie de continuidad entre cada nivel, para que pareciera que todo junto era un súper nivel, teniendo el mismo diseño al final de cada nivel y en el principio del siguiente como se observa en la figura 6.53.

Finalmente viene algo que, sin duda, ha sido aquello que se ha llevado el mayor tiempo de toda la iteración: El remodelado del movimiento de los enemigos. Ya era hora que cambiara el comportamiento debido a un problema que iba arrastrando durante algún tiempo: Al existir la posibilidad que hayan varios enemigos actualizándose a la vez en pantalla, la rutina de comprobación de colisiones de éstos, puesto que es algo costosa, está limitada a que se ejecute una vez cada 16 píxeles, o lo que es lo mismo un bloque entero.

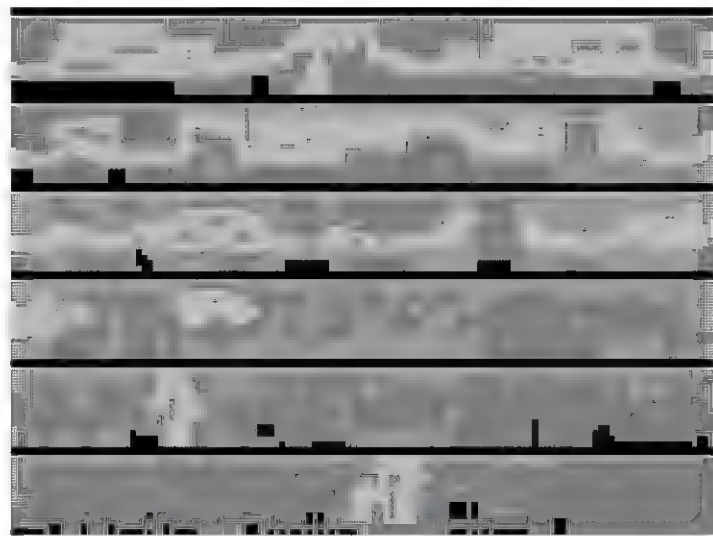


Figura 6.52: Todos los niveles disponibles para jugar

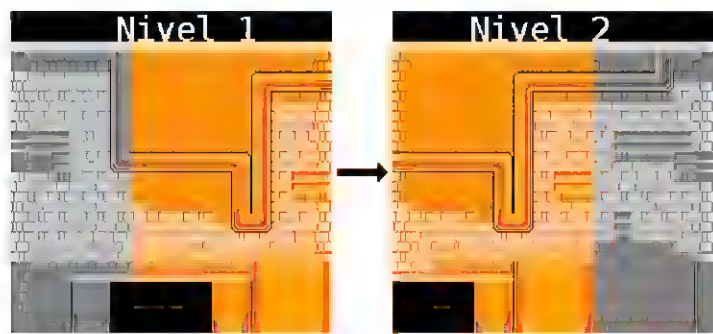


Figura 6.53: Continuidad entre niveles

La forma en la que los enemigos se movían era la siguiente: Se obtenía el valor del desplazamiento en el frame actual y el incremento del scroll, en el caso que hubiera scroll en ese frame, y se sumaba tal cual al valor actual de la posición de cada enemigo. Como se muestra en la tabla 6.10 había algunas situaciones en las que, si el valor final pasaba de ser múltiplo de 0x0F, el personaje seguía su curso y no colisionaba cuando lo tenía que hacer, causando que la progresión de todo el nivel se destrozara por completo, ya que un enemigo se colocaba en una posición que no lo tenía que hacer.

Valor anterior	Valor scroll	Valor desplazamiento	Valor final	¿Múltiplo?
0F	01	00	10	Sí
0F	01	01	11	No

Tabla 6.10: Error con el antiguo sistema de movimiento de los enemigos

Por ello el sistema de movimiento de los enemigos ha sido cambiado para evitar el error nombrado anteriormente: Como se observa en la figura 6.54, ahora mismo añade

el valor que tenga que añadir comprobando, siempre por separado, que dicha condición se cumpla. Si se cumple la condición, en el siguiente frame entra a la rutina de comprobación de las colisiones. Es más, esta comprobación no hace falta hacerla en todas las iteraciones del bucle que desplaza los sprites del enemigo, con hacerlo en la primera iteración es más que suficiente.

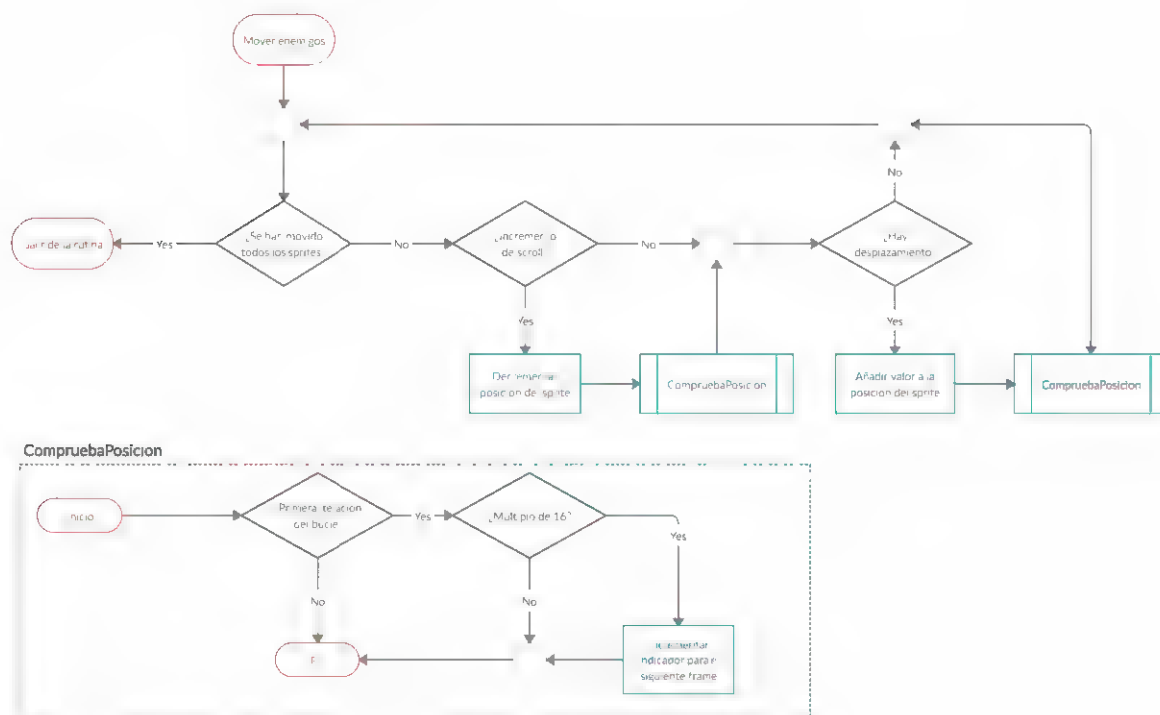


Figura 6.54: Diagrama que explica las comprobaciones sobre el nuevo sistema de movimiento de los enemigos

La segunda parte de esta remodelación venía dada por el hecho que los enemigos tienen un lapso de vida muy corto: Ahora mismo en cuanto éstos llegan a alguno de los extremos de la pantalla, se eliminan automáticamente. En algunos videojuegos analizados en el capítulo 2, como el Mario Bros, los enemigos tienen un tiempo de vida mucho más largo con el concepto de "vivir más allá de la pantalla principal".

La solución final viene dada por la adición de unos offset a la hora de calcular las colisiones, ya que al fin y al cabo es la única rutina que frena al enemigo de moverse para un lado u otro. Como se observa en la figura 6.55, si el enemigo va a comprobar las colisiones en la pantalla principal, no habrá que añadir nada nuevo, en cambio si va a comprobar las colisiones fuera de la pantalla principal habrá que añadir el offset que se indica. En mi caso este offset es el número de columnas que hay de una pantalla hasta otra, es decir 16.

Una de las cosas que más me ha costado determinar son las bandas críticas que se muestran en colores en la misma figura anterior (6.55): Éstas indican cual es el offset que debe utilizarse la próxima vez que se entre a la rutina de comprobación colisiones

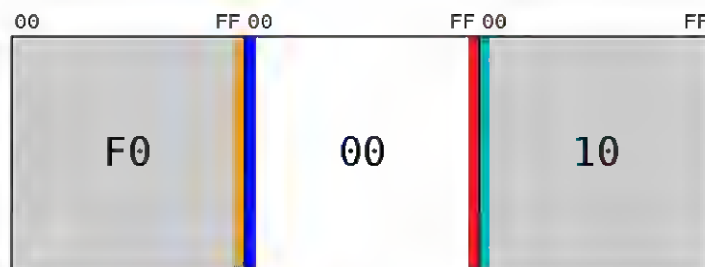


Figura 6.55: Offset que se añaden a la rutina de comprobación de colisiones

y vienen dadas por las situaciones que se indican en la tabla 6.11.

Movimiento	Offset actual	Offset siguiente
Izquierda	0x10	0x00
Izquierda	0x00	0xF0
Derecha	0xF0	0x00
Derecha	0x00	0x10

Tabla 6.11: Offset aplicados al llegar a las bandas críticas de la pantalla

Hay una situación especial que elimina cualquier cambio que se haya realizado al llegar a una de las bandas críticas, aunque solo se aplica cuando el enemigo está en la pantalla principal, por así decirlo, y en uno de los bordes: Tal y como se muestra en la figura 6.56, si el enemigo está apunto de cruzar una de las bandas críticas pero unos bloques más adelante se encuentra en la posición de tener que cambiar su dirección de movimiento, el offset aplicado se restablece totalmente a 00 y todo se vuelve a ejecutar como si no hubiera pasado nada.

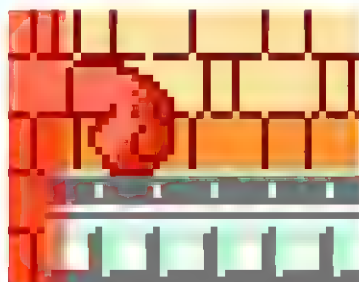


Figura 6.56: Situación especial a la hora de aplicar los offset a los enemigos

Una vez implementé este sistema, solo quedaba esconder los sprites, para que no aparecieran en los lados contrarios cuando cruzaran las bandas críticas. Esta "desactivación" se hace de forma separada según la posición en X de los sprites de la parte izquierda o derecha del enemigo y las bandas críticas: Unas condiciones para las bandas críticas de la izquierda y otras para las de la derecha, como se observa en la tabla 6.12, según se mueva hacia la izquierda o derecha.

Dirección de movimiento	Condición
Izquierda	$x < 0$
Derecha	$x-8 \geq 0$

Tabla 6.12: Condiciones para esconder los sprites de los enemigos cuando no deben aparecer en la pantalla

6.13.8. Iteración 8 - Puliendo las transiciones

- **Fecha de inicio:** 4 de mayo de 2020
- **Fecha de finalización:** 17 de mayo de 2020
- **Duración:** 14 días
- **Descripción:**

Me encuentro ya en la última iteración de todo el proyecto y, aunque indiqué que la iteración 6 era la última en la que iba a añadir más contenido al videojuego, no es así. Realmente no es añadir más contenido de que ya hay al videojuego, sino dejar un juego más limpio y pulido. Por ello, he añadido tres características nuevas: Una pantalla de transición entre los niveles, una animación de muerte para el personaje y una pantalla final con texto.

Como se puede observar en estos tres añadidos nuevos, al final son estados intermedios que son ejecutados entre medias de los niveles o cuando pasan ciertas acciones, por lo que todos ellos tienen sus propios estados con las rutinas de actualización necesarias para que sean ejecutados y no se ocupe mucha más memoria de lo necesario, además de no cargar el estado principal del juego de rutinas y condiciones que no van a actuar normalmente.

Primeramente, la pantalla de transición entre niveles no ha sido gran cosa en cuanto a complejidad. Mi primera idea era que fuera una especie de transición de zonas para evitar "cortes" entre los niveles. Finalmente, y debido al poco tiempo que me quedaba, no podía dedicar mucho más tiempo al proyecto, por lo que al final se realiza un barrido de la pantalla, como se muestra en la figura 6.57: Simplemente se van rellenando todas las columnas de izquierda a derecha hasta que la pantalla es totalmente negra y después se pinta como siempre el nivel siguiente.

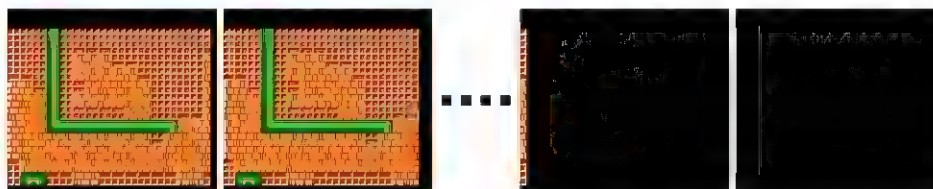


Figura 6.57: Transición de paso entre niveles

La animación de muerte del personaje, ha sido una de las decisiones más importantes debido al hecho de que debes transmitir al usuario que efectivamente ha muerto y se

debe reiniciar el nivel de forma bastante clara, por lo que no puede durar 0.1 segundos. El resultado final consta de dos partes: El cambio de las paletas y el parpadeo de la paleta que utiliza el personaje.

Por una parte, cambio todos los colores de las 8 paletas disponibles a una escala de grises para que toda la pantalla se vea en blanco y negro en el momento de la muerte, como se observa en la figura 6.58.

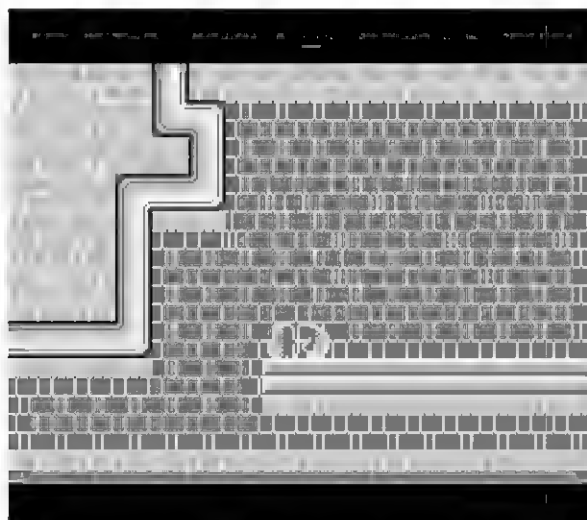


Figura 6.58: Pantalla en blanco y negro por la muerte del personaje

Por último, cada varios ciclos se cambia la paleta de colores que utiliza el personaje principal, para dar más sensación de que realmente acaba de pasar algo, como se observa en la figura 6.59.



Figura 6.59: Cambio de la paleta de colores del personaje en su muerte

Finalmente, y no menos importante, una vez que el jugador termina el videojuego aparece una pantalla que muestra un pequeño texto. En un principio este tipo de pantallas

se podría hacer simplemente creándola como cualquier otra pantalla de cualquier nivel, pero esto hubiera ocupado más memoria de lo necesario. Por ello he decidido crear una pequeña rutina para escribir texto sin la utilización de sprites, es decir, en la parte del fondo.

Como no quería utilizar otro banco más de la PPU para crear otra pattern table que contuviera todas las letras del abecedario, decidí lo siguiente: Este estado utilizaría como tiles del fondo las que normalmente se usarían para los sprites, dado que esta zona en el set de tiles que utiliza el inicio del juego está totalmente sin utilizar, como se observa en la figura 6.60.



Figura 6.60: Letras del abecedario en la pattern table

Para utilizar la rutina que se encarga de escribir las palabras, hay que seguir los siguientes pasos:

1. Aunque sea por comodidad, definir constantes que tengan el valor de cada uno de los índices que utiliza cada letra. De esta forma el pintado de palabras es mucho más fácil y menos ambigua como se observa en la figura 6.61.

```
L_PABLO_SIN:
    .db $11,$01,$02,$0D,$10, $FF

L_PABLO_CON:
    .db C_P,C_A,C_B,C_L,C_O, $FF
```

Figura 6.61: Comparación de escritura de palabras con y sin constantes definidas

2. A continuación se establecen las coordenadas de pintado: La Y será el primer valor que se establezca en el primer acceso a la dirección 0x2006 y la X se utilizará en el segundo acceso al mismo registro.
3. Para terminar, a través de un bucle se recorre la palabra en cuestión hasta conseguir

el valor `0xFF` que indica el final de ésta. La construcción de la pantalla final se observa en la figura 6.62.

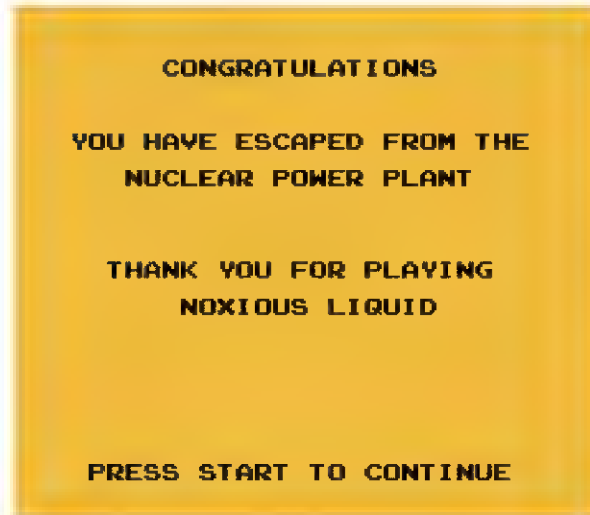


Figura 6.62: Pantalla final con texto

7. Conclusiones

Sin duda, este ha sido el proyecto más ambicioso que he creado hasta ahora y aunque lo mejor de todo es que **para mí ha sido como un juego**, ha acabado pasando por diversas etapas.

A toro pasado, el inicio del proyecto fue muy lento y estaba muy confiado sabiendo que tenía por delante "muchos meses" para acabar el desarrollo. Mal por mi parte, puesto que podría haber incluido más contenido en el producto final. De todas formas, mi objetivo desde un principio fue realizar un videojuego del que pudiera estar orgulloso y que pudiera abrirme algunas puertas en el futuro y no estar trabajando en algo que hay que hacerlo porque sí.

Claro está, he tenido muchos momentos en los que la moral la tenía por los suelos, de pensar que no iba a terminar el proyecto de la forma que quería finalizarlo puesto que no conseguía aplicar aquello en lo que estaba trabajando y pensando horas y horas. Estos momentos me han llevado a una forma totalmente distinta de pensar y trabajar, ya que **una mente descansada trabaja mucho mejor**: De estar horas seguidas intentando algo, dejarlo apartado por un día entero, sin pensar en ello, para que al día siguiente consiga solucionarlo en 10 minutos de reloj.

Para terminar, estoy totalmente alucinado con el resultado final del videojuego. A excepciones que se me han ido ocurriendo en su etapa final, y que ya era demasiado tarde para ponerme a trabajar, el producto final está a las expectativas de aquello que desde un principio tenía en mi cabeza siendo, lo más importante, que es un videojuego con un principio y un final:

- He creado una guía de desarrollo para aquellos desarrolladores que quieran realizar videojuegos para la NES.
- Me he adaptado a las condiciones de desarrollo del sistema, debido a sus escasas especificaciones, al haber realizado un juego totalmente optimizado para ejecutarse en NTSC y PAL y haber aprovechado al máximo la memoria que la ROM dispone.
- He sido capaz de crear un diseño visual con personajes, animaciones, niveles, pantallas y colores totalmente propios.

Creo a ciencia cierta, que este sistema ha ganado un nuevo desarrollador y de ninguna manera quiero desperdiciar todo el conocimiento que he adquirido a lo largo de estos meses, puesto que mi objetivo es mejorar Noxious Liquid para convertirlo en un mejor videojuego.

A. Lenguaje ensamblador 6502

A.1. Set de instrucciones del lenguaje ensamblador 6502

El MOS 6502 tiene un total de 56 instrucciones de memoria.

Tabla A.1: Set de instrucciones del MOS Technology 6502

Instrucción	Explicación
ADC	Añade un valor al Acumulador añadiendo el valor del flag Carry
AND	Operación AND del Acumulador con un valor
ASL	Desplazamiento hacia la izquierda de los bits en 1
BCC	Ramificación si el flag Carry está inactivo
BCS	Ramificación si el flag Carry está activo
BEQ	Ramificación si el flag Zero está activo
BIT	Test Bits in Memory with Accumulator
BMI	Ramificación si el flag Sign está activo
BNE	Ramificación si el flag Zero está inactivo
BPL	Ramificación si el flag Sign está inactivo
BRK	Forzar una interrupción BRK
BVC	Ramificación si el flag Overflow está desactivado
BVS	Ramificación si el flag Overflow está activo
CLC	Desactivar el flag Carry
CLD	Desactivar el modo decimal
CLI	Desactivar el flag Interrupt
CLV	Desactivar el flag Overflow
CMP	Comparación de un valor con el Acumulador
CPX	Comparación de un valor con el registro X
CPY	Comparación de un valor con el registro Y
DEC	Decremento en 1 del valor almacenado en una dirección de memoria
DEX	Decremento en 1 del valor almacenado en el registro X
DEY	Decremento en 1 del valor almacenado en el registro Y
EOR	Operación XOR del Acumulador con un valor
INC	Incremento del Acumulador en 1
INX	Incremento del registro X en 1
INY	Incremento del registro Y en 1
JMP	Salto a una dirección de memoria
JSR	Salto a una dirección de memoria, guardando la dirección de retorno
LDA	Cargar un valor en el Acumulador
LDX	Cargar un valor en el registro X
LDY	Cargar un valor en el registro Y

Continuación en la siguiente página

Tabla A.1 – *Continuación de la anterior página*

Instrucción	Explicación
LSR	Desplazamiento hacia la izquierda de los bits en 1
NOP	No se realiza operación (No OPeration)
ORA	Operación OR del Actumulador con un valor
PHA	Push del valor del Acumulador en la pila
PHP	Push del estado de proceso a la pila
PLA	Pull de la pila al Acumulador
PLP	Pull del estado de proceso a la pila
ROL	Rotación hacia la izquierda de los bits en 1
ROR	Rotación hacia la derecha de los bits en 1
RTI	Retorno/vuelta a una interrupción
RTS	Retorno/vuelta a una subrutina
SBC	Sustraer un valor al Acumulador añadiendo el valor del flag Carry
SEC	Activar el flag Carry
SED	Activar el modo decimal
SEI	Activar el flag Interrupt
STA	Almacenar el valor del Acumulador en memoria
STX	Almacenar el valor del registro X en memoria
STY	Almacenar el valor del registro Y en memoria
TAX	Transferencia del valor del Acumulador al registro X
TAY	Transferencia del valor del Acumulador al registro Y
TSX	Transferencia del valor del puntero de la pila al registro X
TXA	Transferencia del valor del registro X al Acumulador
TXS	Transferencia del valor del registro X al puntero de la pila
TYA	Transferencia del valor del registro X al Acumulador

A.2. Registros

Los registros, excepto el de estados, son utilizados para copiar contenidos de la memoria a éstos y viceversa. El MOS 6502 dispone de cuatro registros en total.

A.2.1. Acumulador

El acumulador (A) es el registro más importante de todos: Es el único sobre el que se pueden realizar operaciones matemáticas.

Código A.1: Ejemplo de instrucciones sobre el registro A

```

LDA #FF ;; Carga del valor 0xFF en A
ADC $C000 ;; A + Valor que hay en 0xC000
AND $C000 ;; Operación AND de A con el valor que hay en 0xC000
ASL ;; Bit shifting hacia la izquierda

```

A.2.2. El registro X e Y

Ambos registros son casi idénticos. La diferencia entre ellos es la forma en que son utilizados dentro del microprocesador, ya que hay algunas instrucciones y modos de memoria que no se pueden llevar a cabo con uno, pero con otro sí y viceversa.

Código A.2: Ejemplo de instrucciones sobre los registros X e Y

```
LDX #$FF ;; Carga de valor 0xFF en X
LDY #$FF ;; Carga el valor 0xFF en Y

TAY      ;; Transfiere el valor de A a Y
TAX      ;;

TSX      ;; Transfiere el valor al que apunta el puntero de la pila a X
;; Como comentaba antes, esta instrucción NO puede llevarse a cabo con el registro Y
```

A.2.3. El registro de estados

Un registro compuesto por ocho bits, como se observa en la tabla A.2, que se encargan de indicar aquellos cambios que se han producido en operaciones lógicas y aritméticas. Dependiendo del resultado de estas operaciones, ciertos bits, o flags, son alterados.

Bit	Flag	Activación
0	Carry	Mantiene el bit más significativo de acarreo de cualquier operación aritmética
1	Zero	Se convierte en 1 en el momento que cualquier operación aritmética produce un resultado igual a cero
2	Interrupt	Si está activa, las interrupciones están desactivadas
3	Decimal	Modo decimal para el sistema
4	Break	Se activa en el momento que una interrupción BRK es ejecutada
5	—	No se utiliza. Siempre es 1
6	Overflow	Se activa en el momento que una operación produce un resultado demasiado grande para representarlo con un byte
7	Sign	El bit que indica el signo de una operación aritmética. Si se activa, el resultado de la operación habrá sido negativo

Tabla A.2: Flag disponibles

A.2.4. El contador del programa

Contiene la dirección de memoria de la instrucción actual que está siendo ejecutada. No es posible acceder a este registro mediante las instrucciones, aunque es actualizado constantemente debido a instrucciones de salto (JMP), ramificaciones (BEQ), llamada a subrutinas (JSR) o interrupciones.

A.2.5. El puntero a la pila

Contiene la dirección de memoria del primer hueco libre dentro de la pila. Ésta es un almacenamiento temporal, que se puede utilizar libremente con diversas instrucciones del microprocesador.

Código A.3: Ejemplo de instrucciones que involucren el uso de la pila

```
PHA ;; Push del valor en el acumulador en la pila
PLA ;; Pull del valor en la pila al acumulador

JSR $C000 ;; Salta a la dirección de memoria 0xC000 y guarda en la pila la dirección de retorno
RTS      ;; Recoge la dirección de memoria almacenada de la pila y el contador de programa toma su ←
          ↪ valor
```

A.3. Modos de direccionamiento a la memoria

Antes de empezar la explicación, quisiera recalcar que aquí utilizo una notación específica a la hora de explicar los modos de direccionamiento. Esto quiere decir que dependiendo del compilador que se utilice, éste tendrá una forma u otra de definir estos conceptos/modos.

A.3.1. Inmediato

Este modo, como su nombre indica, es totalmente inmediato, es decir, el propio programador puede indicar un valor directamente sobre el código, junto a la instrucción, para cargarlo en un registro.

Código A.4: Modo de direccionamiento inmediato

```
LDX #$FF ;; Carga de valor 0xFF en X
LDY #$FF ;; Carga el valor 0xFF en Y
LDA #$FF ;; Carga el valor 0xFF en A
```

A.3.2. Zero-page

Se define como zero-page aquella porción de memoria donde su byte alto siempre es 00, es decir, todas las direcciones de memoria que van desde 0x0000 a 0x00FF. Los accesos a zero-page siempre serán mucho más rápidos que el acceso a cualquier otra dirección de memoria.

En algunos de los siguientes modos, se explican algunas variaciones que tienen en el caso que el acceso sea a zero-page.

A.3.3. Absoluto

El acceso a la memoria del modo absoluto se hace de tal forma que se le indica al procesador la dirección de memoria donde se encuentra el valor que se requiere.

Código A.5: Modo de direccionamiento absoluto

```
;; Pongamos que en 0xC000 se encuentra el valor 0xFF
LDA $C000 ;; Carga el valor que se encuentra en 0xC000 en A.
```

El modo absoluto tiene una variante respecto a la zero-page que se define con la siguiente instrucción. Nótese que no se utiliza la notación de siempre, sino que se omite el # a la hora de indicar el operando, ya que no es un valor en hexadecimal, es una dirección de memoria.

Código A.6: Modo de direccionamiento absoluto zero-paged

```
;; Pongamos que en 0x00C0 se encuentra el valor 0xFF
LDA $C0 ;; Carga el valor que se encuentra en 0x00C0 en A.
```

A.3.4. Implícito

Las instrucciones que se incluyen en este modo, no necesitan operadores de dirección. Como su nombre indica, están implícitos en la instrucción.

Código A.7: Modo de direccionamiento implícito

```
TAX ;; Transferencia del valor del registro A al registro X
INC ;; Incremento del registro A en 1
INX ;; Incremento del registro X en 1
```

A.3.5. De acumulador

Como señala el título, aquí se encuentran instrucciones que sólo realizan operaciones sobre el acumulador. Al igual que el anterior, no se necesitan operaciones de dirección a la hora de escribir las instrucciones.

Código A.8: Modo de direccionamiento de acumulador

```
LSR ;; Bit shifting hacia la izquierda
INC ;; Incremento del registro A en 1
ROR ;; Rotación de bits en 1 hacia la derecha
```

A.3.6. Indexado

En este modo, se añade un operador de dirección de más, a parte de los que se han visto hasta ahora. El valor que se almacene en X o Y se va a añadir al valor al que se accede mediante el modo absoluto.

Código A.9: Modo de direccionamiento indexado

```
LDX #$00 ;; Almacenamos el valor 0x00 en el registro X
STX $C000 ;; Guardamos el valor de X en 0xC000

LDY #$10 ;; Almacenamos el valor 0x10 en el registro Y
LDA $C000, Y ;; En A es cargado el valor que reside en 0xC000 (0x00) y se le añade el de Y (0x10)
```

También tiene un modo zero-paged, aunque solo es posible utilizarlo junto al registro X.

Código A.10: Modo de direccionamiento indexado zero-paged

```
LDY #$00 ;; Almacenamos el valor 0x00 en el registro Y
STY $00C0 ;; Guardamos el valor de Y en 0xC000

LDX #$10 ;; Almacenamos el valor 0x10 en el registro X
LDA $C0, X ;; En A es cargado el valor que reside en 0xC000 (0x00) y se le añade el de X (0x10)
```

A.3.7. Indirecto pre-indexado

En este modo, se indica como primer operando una dirección de memoria y como segundo operando el valor de desplazamiento respecto al primer operando, accediendo, finalmente, a la dirección de memoria que almacena esta suma y obtener el valor que almacena.

Es muy importante saber que el modo de direccionamiento pre-indexado solo puede utilizarse con direcciones de memoria que apunten a la zero-page y, este en concreto, sólo junto al registro X como segundo operando.

Código A.11: Modo de direccionamiento indirecto pre-indexado.

```
;; Pongamos que en 0x0010 se encuentra un valor de 0xC0
LDY #$C0
STY $0010

;; En 0x0011 se encuentra un valor de 0x00
LDY #$00
STY $0011

;; Y, finalmente, en 0x00C0 se encuentra un valor de 0xFF
LDY #$FF
STY $00C0

;; 0010    C0 00 ..
;; ....    .. .. ..
;; 00C0    FF ..

LDX #$10 ;; El "desplazamiento" será de 0x10
LDA ($00, X)

;; El primer operando es la dirección de memoria inicial, en este caso 0x0000
;; Añadimos el valor del desplazamiento, quedando como 0x0010
;; Ahora se observa la dirección de memoria que hay almacenada en la dirección 0x0010
;; [0x0010] C0
;; [0x0011] 00
;; Está almacenada la dirección de memoria 0x00C0
;; Finalmente se accede al valor que tiene esta última dirección de memoria
;; A = 0xFF
```

A.3.8. Indirecto post-indexado

Al igual que el pre-indexado, este modo solo puede utilizarse junto a direcciones de memoria que apunten a la zero-page y, a diferencia del anterior, éste solo puede utilizarse junto al registro Y.

Indicando una dirección de memoria de la zero-page, como primer operando, y un valor en el registro Y, como segundo operando, se desenvuelve de forma similar al pre-indexado. La diferencia es cuando se realiza la suma.

Código A.12: Modo de direccionamiento indirecto post-indexado.

```
;; Pongamos que en 0x0010 se encuentra un valor de 0xB0
LDY #$B0
STY $0010

;; En 0x0011 se encuentra un valor de 0x00
```

```
LDY #$00
STY $0011

;; Y, finalmente, en 0x00C0 se encuentra un valor de 0xFF
LDY #$FF
STY $00C0

LDY #$10 ;; Al igual que el anterior, el "desplazamiento" será de 0x10
LDA ($00, Y)

;; Se observa la dirección de memoria que es pasada como operando, 0x0000
;; Ahora se obtiene la dirección de memoria que hay almacenada en la dirección 0x0010
;; [0x0010] B0
;; [0x0011] 00
;; Está almacenada la dirección de memoria 0x00B0
;; Añadimos el valor del desplazamiento, quedando como 0x00C0
;; Finalmente se accede al valor que tiene esta última dirección de memoria
;; A = 0xFF
```

A.3.9. Indirecto

Este modo solamente se aplica a la instrucción JMP, que sirve para saltar (JuMP) a una nueva ubicación. En este caso, el operador utilizará los dos valores que guarda la dirección de memoria que hay entre paréntesis.

Código A.13: Modo de direccionamiento indirecto

```
JMP $C000 ;; Salta a la dirección de memoria 0xC000

JMP ($C000) ;; Salta a la dirección de memoria almacenada en las direcciones 0xC000 y 0xC001
;; [0xC000] 34
;; [0xC001] 12
;; Es decir, la instrucción JMP saltaría a la dirección de memoria 0x1234
```

A.3.10. Relativo

Las instrucciones que utilizan el registro de estados como condición, son aquellas que utilizan el modo de direccionamiento relativo. Para ello, se encargan de observar el estado del bit correcto.

Código A.14: Modo de direccionamiento relativo

```
LDA #$EF ;; Cargamos el valor 0xEF en el registro A
ADC #$11 ;; Añadimos al registro A el valor 0x11

;; Se activa el flag Zero puesto que el resultado ha dado 0x00
;; También se ha activado el flag Carry ya que se "ha dado una vuelta al byte"

BEQ $D000 ;; Branch Equal; saltaríamos a la dirección de memoria 0xD000
BCS $D000 ;; Branch Carry Set; también saltaríamos a la dirección de memoria 0xD000

;; ADC es una instrucción que añade un valor al que hay en el acumulador
;; También se encarga de añadir el valor del flag Carry del registro de estados
;; Por lo tanto, si al ejecutar esta instrucción el flag Carry está activo, añade uno más
```


B. CPU

B.1. RAM

La RAM, es aquella memoria modificable mientras en programa está en ejecución. Ocupa un total de 2 kb de memoria: desde la dirección 0x0000 a la 0x0800. Además, es cuadruplicada hasta la dirección de memoria 0x1FFF.

Las primeras 256 direcciones corresponden a la sección de la zero-page de la RAM, proporcionando accesos más rápidos con la ventaja de utilizar direccionamientos especiales de la memoria.

B.2. Pila

En la NES la pila ocupa un total de 256 bytes de memoria, desde la dirección 0x0100 hasta la 0x01FF. Funciona de arriba hacia abajo, es decir, el primer valor almacenado en la pila siempre será en 0x01FF, el segundo en 0x01FE y así sucesivamente.

Hay que llevar mucho cuidado a la hora de tratar con la pila, puesto que la CPU no tiene control a la hora de advertir si se ha superado el límite de memoria almacenada en la sección de la pila. En esta situación, al ser, la memoria de la CPU, lineal, continuaría por la sección de memoria que corresponde a la zero-page, pudiendo llegar a sobrescribir los datos que hay almacenados en esta última.

B.3. RAM interna

A partir de la dirección de memoria 0x0200, hasta 0x07FF, se encuentra la RAM interna, dentro del bloque que posee su mismo nombre. Esta porción de memoria puede ser libremente utilizada para el almacenamiento de datos, aunque teniendo en cuenta lo siguiente: No pasarse de la memoria asignada y tener en cuenta la zona de sprites.

Zona de sprites: Los primeros 256 bytes de memoria corresponden a la zona donde se almacenan los datos de los sprites. En total pueden almacenarse 64 sprites siguiendo el esquema de memoria indicado en la tabla B.1.

B.4. Registros de entrada y salida

Se encargan de la comunicación de la CPU con la PPU y la Unidad de Procesamiento de Audio (APU). Están organizados en memoria a partir de la dirección de memoria 0x2000,

Sprite	Dirección	Información sobre sprite
Sprite 1	0x0200	Posición en Y
	0x0201	Número de sprite en la sprite pattern
	0x0202	Color y atributos: <i>bit 0,1</i> : Set de 4 colores a usar en el sprite <i>bit 2,3,4</i> : No se conoce uso <i>bit 5</i> : Prioridad, 0 = delante del fondo <i>bit 6</i> : Voltar el sprite horizontalmente <i>bit 7</i> : Voltar el sprite verticalmente
	0x0203	Posición en X
Sprite 2	0x0204	Posición en Y
.....

Tabla B.1: Organización de la memoria asignada a los sprites en la CPU

cuando termina la memoria dedicada a la RAM y sigue el esquema de memoria que se explica en la tabla B.2.

Tabla B.2: Organización de la memoria asignada a los registros de entrada y salida en la CPU

Dirección	Permisos	Bit	Información
0x2000	Lectura Escritura		Registro de control 1 de la PPU
		<i>bit 0,1</i>	Selección de nametable - 00 : Selecciona la nametable de 0x2000 - 01 : Selecciona la nametable de 0x2400 - 10 : Selecciona la nametable de 0x2800 - 11 : Selecciona la nametable de 0x2C00
		<i>bit 2</i>	Escritura vertical Cuando toma el valor de 1, la memoria de la PPU aumenta en incrementos de 32
		<i>bit 3</i>	Pattern table para sprites - 0 : Elige la pattern table de 0x0000 - 1 : Elige la pattern table de 0x1000
		<i>bit 4</i>	Pattern table para el fondo - 0 : Elige la pattern table de 0x0000 - 1 : Elige la pattern table de 0x1000
		<i>bit 5</i>	Tamaño de los sprites - 0 : Sprites de tamaño 8x8 píxeles - 1 : Sprites de tamaño 8x16 píxeles
		<i>bit 6</i>	No se utiliza en la NES
		<i>bit 7</i>	Bit de VBlank
0x2001	Lectura Escritura		Registro de control 2 de la PPU
		<i>bit 0</i>	Indica el modo de color del sistema - 0 : Modo color - 1 : Modo monocromo
		<i>bit 1</i>	Indica el recorte del fondo - 0 : Si se quiere que se esconda el fondo 8 píxeles a la izquierda
		<i>bit 2</i>	Indica el recorte de los sprites - 0 : Si se quiere que se escondan los sprites 8 píxeles a la izquierda
		<i>bit 3</i>	Toma el valor de 1 si el fondo debe dibujarse
		<i>bit 4</i>	Toma el valor de 1 si los sprites deben dibujarse
		<i>bit 5,6,7</i>	Indica el color de fondo en cuando está en modo monocromo o la intensidad cuando está activo el modo de color - 000 : No hace nada - 001 : Intensifica el verde - 010 : Intensifica el azul - 100 : Intensifica el rojo
0x2002	Escritura	<i>bit 4</i>	Registro de estado de la PPU Si se activa, ignora cualquier acceso de escritura

Continuación en la siguiente página

Tabla B.2 – Continuación de la anterior página

Dirección	Permisos	Bit	Información
			a la VRAM
		<i>bit 5</i>	Cuenta los sprites de la scanline actual
		<i>bit 6</i>	Activa el bit en el caso que hayan más de 8 sprites
			Indicador de colisión del sprite 0
			Activa el bit en el caso que un píxel no transparente del sprite no superponga un píxel no transparente del fondo
		<i>bit 7</i>	Indica si la VBlank está sucediendo
0x2003	Lectura Escritura		Dirección apuntando a la memoria de sprites Utilizado para establecer una offset de memoria que apunta a la memoria de sprites y por la que, mediante el registro en 0x2004 se accederá Esta dirección será siempre incrementada en 1 después de cada acceso
0x2004	Lectura Escritura		Datos de de la memoria de sprites Accederá a la dirección que está establecida mediante el acceso a 0x2003
0x2005	Lectura		Offset del desplazamiento de la pantalla Indica el valor que se aplicará como offset La primera escritura pertenece al registro de desplazamiento vertical, en el caso que sea siempre menor o igual que 239 La segunda escritura pertenece al registro de desplazamiento horizontal
0x2006	Lectura		Dirección de memoria de la PPU Utilizado para establecer una dirección de memoria de la PPU Será accesible mediante la dirección 0x2007, incrementando en 1, o 32, en cada acceso a ésta, según el bit 2 de 0x2000 La primera escritura establecerá los 8 bits bajos y la segunda 6 bits altos
0x2007	Lectura Escritura		Datos de memoria de la PPU
.....			
Mirrors de las direcciones anteriores hasta 0x3FFF			
.....			
0x4000	Escritura		Registro relacionado con el sonido
0x4001	Escritura		Registro relacionado con el sonido
0x4002	Escritura		Registro relacionado con el sonido
0x4003	Escritura		Registro relacionado con el sonido

Continuación en la siguiente página

Tabla B.2 – Continuación de la anterior página

Dirección	Permisos	Bit	Información
0x4004	Escritura		Registro relacionado con el sonido
0x4005	Escritura		Registro relacionado con el sonido
0x4006	Escritura		Registro relacionado con el sonido
0x4007	Escritura		Registro relacionado con el sonido
0x4008	Escritura		Registro relacionado con el sonido
0x4009	Escritura		Registro relacionado con el sonido
0x400F	Escritura		Registro relacionado con el sonido
0x4010	Escritura		Registro relacionado con el sonido
0x4011	Escritura		Registro relacionado con el sonido
0x4012	Escritura		Registro relacionado con el sonido
0x4013	Escritura		Registro relacionado con el sonido
0x4014	Escritura		Acceso directo a la memoria de sprites La escritura de un valor N en este registro provocará la transferencia de un total de 256 bytes a la memoria de sprites localizada en la PPU, apuntando a la dirección en la CPU determinada por $0x100 \cdot N$ Es decir, si N es el valor 0x02, la transferencia de memoria se realizará desde 0x0200 en la CPU
0x4015	Escritura		Registro relacionado con el sonido
0x4016	Lectura		Controles del jugador 1
	Escritura	bit 0	Lectura de los datos del mando 1
		bit 3	Indica si el Zapper apunta a un sprite
		bit 4	0 : Cuando el gatillo del Zapper ya no está pulsado
0x4017	Lectura		Controles del jugador 2
	Escritura		Actúa de la misma forma que el registro anterior

B.5. Controles

En la NES existen dos registros definidos para la lectura de los controles, 0x4016 para el jugador 1 y 0x4017 para el jugador 2. Esta lectura se realiza mediante el acceso a uno de los dos registros nombrados. El primer acceso siempre va a leer si el jugador ha pulsado la A, el segundo la B, y así sucesivamente. En concreto, el orden de acceso es el siguiente:

A > B > SELECT > START > UP > DOWN > LEFT > RIGHT

Código B.1: Funcionamiento de los controles.

```
;; Primero debe realizarse esta escritura en el registro
;; concreto del que queramos conseguir los datos del mando
LDA #$01
STA $4016
LDA #$00
STA $4016

LeerA1:
    LDA $4016 ;; Lectura del mando del jugador 1
```

```
    AND #%00000001 ;; Solo hace falta mirar el bit 0
    BEQ LeerA1Fin
    ;; Aquí podemos poner las acciones que queramos realizar
    ;; cuando el botón A del jugador 1 es pulsado
LeerA1Fin:

LeerB1:
    LDA $4016 ;; Lectura del mando del jugador 1
    AND #%00000001 ;; Solo hace falta mirar el bit 0
    BEQ LeerB1Fin
    ;; Lo mismo que con el botón A
LeerB1Fin:

;; ...
```

B.6. ROM de expansión

Como su nombre indica, es una expansión de la ROM, Read Only Memory, en inglés. Cubre las direcciones de memoria desde 0x4020 hasta 0x5FFF y es utilizado, generalmente, para el mapeo de la ROM principal de la NES.

B.7. SRAM

La "Save-RAM", es otro puerto opcional de expansión, similar a la ROM de expansión. Tiene un total de 8 kb de memoria a su disposición, cubriendo desde 0x6000 hasta 0x7FFF. Además, como se indica, puede llegar a ser parte de la RAM principal.

B.8. ROM

Con un tamaño total de casi 32 kb de memoria, la ROM principal es el espacio de memoria más grande del que dispone la NES. Comienza en la dirección de memoria 0x8000 y termina en 0xFFFF. Normalmente es donde se debería almacenar los datos de, por ejemplo, mapas o enemigos, ya que, como su propio nombre indica, es un espacio de memoria de solo lectura: Durante la ejecución del programa no los podremos modificar.

B.9. Bancos de memoria

La memoria dentro de la NES es dividida en bancos de memoria. Cada banco de memoria puede almacenar un total de 8KB y el total de ellos vienen dados por el mapper que se elija, tanto para memoria ROM como para pattern tables de sprites. En este documento solo se cubre la utilización de dos mappers, aunque existen muchos más que se pueden encontrar en la wiki de NesDev¹.

B.10. Interrupciones

Finalmente, a partir de 0xFFFFA, nos encontramos con los tres vectores de interrupción.

¹<https://wiki.nesdev.com/w/index.php/Mapper>

B.10.0.0.1. Vector Una dirección de 16 bits, o 2 bytes, que especifica una localización en la memoria a la que saltar cuando la interrupción es activada.

IRQ/BRK: Es activado en dos situaciones distintas: Mediante software, utilizando la instrucción BRK, o mediante hardware. Es la interrupción con menor prioridad de las tres.

Dirección de memoria: 0xFFFA

NMI: Es activado mediante el refresco continuo de la pantalla. El tiempo de refresco depende del sistema: Si es PAL serán 50 refrescos/segundo y en el caso de NTSC 60 refrescos/segundo. Tiene una prioridad menor que RESET pero mayor que IRQ/BRK.

Dirección de memoria: 0xFFFC

RESET: Como su nombre indica, es activado en el momento de encendido de la consola, aunque se puede acceder a esta rutina de forma intencionada. Accediendo desde la última forma nombrada, se volverá a cargar todo de nuevo en memoria, con el inconveniente que no se reiniciarán los registros ni se limpiarán tramos de memoria, como ocurre en el encendido de la consola. Es la interrupción con mayor prioridad de las tres.

Dirección de memoria: 0xFFFFE

C. PPU

C.1. Pattern table

La NES tiene dos pattern table: La primera localizada en 0x0000 y la segunda en 0x1000. Éstas almacenan los sprites de 8x8 píxeles que serán dibujados por pantalla. Generalmente cada una se utiliza para un ámbito distinto del juego, la primera almacena los sprites de los personajes y la segunda del fondo y los escenarios.

Código C.1: Forma de activar las pattern table

```
;; Si queremos lo siguiente:
;; bit 3 = 0 : Los sprites utilizarán la pattern table localizada en 0x0000
;; bit 4 = 1 : El fondo utilizará la pattern table localizada en 0x1000

LDA #%10010000 ;; Establecemos los valores
STA $2000      ;; Nos comunicamos con la PPU
```

C.1.0.0.1. Carga de sprites Este proceso se realiza mediante una comunicación directa (DMA) desde la CPU hasta la PPU, generalmente, en el momento que la VBlank está ocurriendo.

Código C.2: Activación de la transferencia de los datos de sprites

```
NMI:
LDA #$00 ;; Establecemos el byte bajo
STA $2003 ;; Lo escribimos en memoria

LDA #$02 ;; Establecemos el byte alto
STA $4014 ;; Comenzamos la transmisión de información
...
```

La escritura del valor 0x02 en la dirección de memoria 0x4014 indica que debe empezar en la dirección de memoria establecida por este valor multiplicado por 0x0100: En este caso $0x02 * 0x0100$, es decir, **la transmisión de memoria empezará desde la dirección 0x0200** a la que se le suma **el valor que hemos establecido anteriormente en 0x2003**. Además se ve que este proceso está inmediatamente después de cuando salta la interrupción NMI.

C.2. Name table

Las name tables, o *tablas de nombres*, son matrices de números que almacenan en memoria los id de los sprites almacenados en las pattern tables. Tienen un tamaño total de 32x30 sprites: Como cada sprite ocupa un total de 8x8 píxeles, cada tabla de nombres tiene un tamaño total de 256x240 píxeles.

Código C.3: Carga de los sprites a las name tables

```

;; Primero hay que realizar una lectura en 0x2002
;; para evitar la escritura de información errónea sobre la PPU

;; Para ello, esta lectura reinicia, por así decirlo,
;; el byte alto/bajo que es escrito en la PPU
LDA $2002

;; La primera name table se encuentra en la dirección de memoria
;; 0x2000 de la PPU, por lo tanto, primero escribimos el byte
;; alto de la dirección
LDA #$20
STA $2006
LDA #$00
STA $2006

;; Ya le hemos dicho a donde queremos que cargue los datos
;; Ahora hay que empezar la carga de sprites
LDA #$00 ;; Sprite 0 = 0x00
STA $2007 ;; Se lo pasamos a la PPU con la escritura del valor del sprite en la dirección 0x2007

LDA #$B5 ;; Sprite 1 = 0xB5
STA $2007 ;; Transferencia a la PPU

;; ...
;; Y así sucesivamente hasta que carguemos los 32x30 sprites en la PPU

```

C.2.0.0.1. PAL vs NTSC En el caso que el formato de televisión de la NES sea PAL estos datos se cumplen y no existen recortes en cuanto se habla de líneas de pantalla. En cambio, con el formato NTSC se observa un "recorte" de ocho píxeles en la parte superior de la pantalla y de otros ocho píxeles en la parte inferior, adquiriendo una resolución de 256x224 píxeles.

C.2.0.0.2. Mirroring Aunque la PPU soporta hasta cuatro name tables en total, por defecto no hay suficiente VRAM para las cuatro, por ello dos de ellas siempre van a ser un mirror de las otras dos, como se observa en la figura C.1.

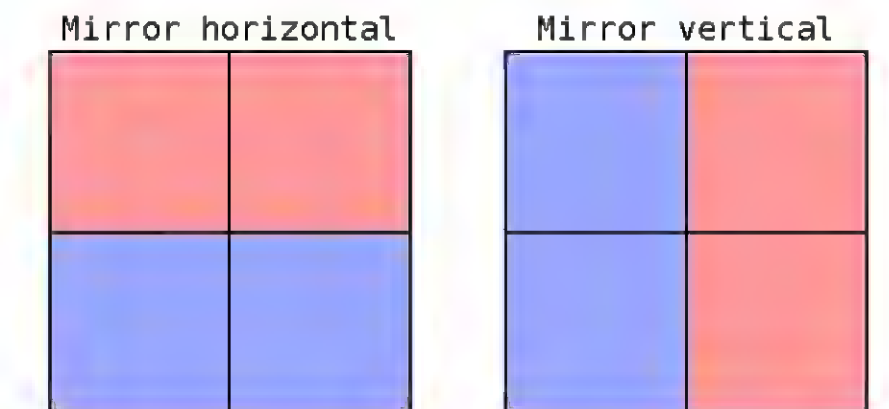


Figura C.1: Representación de los mirror que se realizan dentro de la PPU

Mirror horizontal: La name table situada en 0x2400 será un mirror total de la situada en 0x2000 y la que está situada en 0x2C00 será de 0x2800. Permite el scroll vertical.

Mirror vertical: La name table situada en 0x2800 será un mirror total de la situada en 0x2000 y la que está situada en 0x2C00 será de 0x2400. Permite el scroll horizontal.

C.3. Attribute Table

Toda name table va acompañada en memoria de una attribute table: Éstas se encargan de decirle que colores de la paleta hay que usar. Al igual que con las name tables, hay cuatro attribute tables en memoria de la PPU y ocupan un total de 64 bytes de memoria.

Hay que tener en cuenta que los sprites no son pintados siguiendo un orden en memoria por así decirlo, sino que sigue el siguiente esquema:

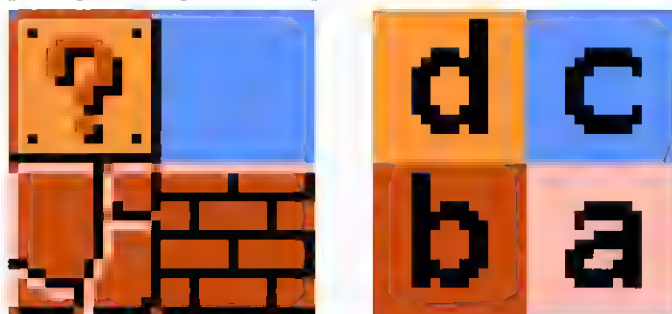


Figura C.2: Orden de pintado de los sprites del fondo

Como se aprecia en la imagen, las letras **a**, **b**, **c** y **d** indican el orden de pintado de los sprites que forman parte del fondo, siendo éstas un valor de dos bits, además de representar, cada una, la agrupación de cuatro sprites de 8x8 píxeles. El valor de estos dos bits viene dado según las paletas de colores que establezcamos en la PPU.

C.3.0.0.1. Paletas de colores En la PPU existen dos zonas dedicadas para las paletas de colores: La primera, la paleta dedicada para el fondo, se encuentra en 0x3F00 y la segunda, dedicada para los colores de los sprites, situada en 0x3F10. Ambas ocupan 16 bytes cada una, o 16 colores. En total, podemos elegir entre un total de 64 colores disponibles.



Figura C.3: Agrupación de los colores en las dos paletas

Como se ve en C.3, están representados como si estuvieran cargados en memoria. Además se observa que están agrupados en grupos de cuatro. Este "índice" es usado cuando queremos indicar, por ejemplo, que cierto sprite use el color 00. Por lo tanto dicho sprite utilizará el grupo de cuatro colores compuesto por 22, 16, 27 y 18.

Código C4: Carga de las attribute table

```

;; Al igual que con la carga de la name table,
;; primero hay que realizar una lectura en 0x2002
;; para evitar la escritura de información errónea sobre la PPU
LDA $2002

;; La primera attribute table se encuentra en la
;; dirección de memoria 0x23C0 de la PPU
LDA #$23
STA $2006
LDA #$C0
STA $2006

LDA #%00000000
;; AA BB CC DD
;; 00 00 00 00
;; -----
;; | DD | CC
;; | -- | --
;; | BB | AA |
STA $2007 ;; Transferencia a la PPU

LDA #%01010000
;; AA BB CC DD
;; 01 01 00 00
;; -----
;; | DD | CC
;; | -- | --
;; | BB | AA |
STA $2007 ;; Transferencia a la PPU

;; ...
;; Y así sucesivamente hasta que carguemos los 64 bytes en la PPU

```

D. Documento de Diseño del Videojuego

D.1. Descripción inicial

- **Autor:** Pablo Máñez Fernández
- **Título:** Noxious Liquid
- **Versión actual:** 1.2
- **Año:** 2019-2020

D.2. Control de cambios

Nº revisión	Descripción	Fecha	Versión
001	Versión inicial con la descripción del primer prototipo realizado	24/03/20	1.0
002	Actualización segundo prototipo con progresión de niveles	20/04/20	1.1
003	Actualización de las mecánicas y objetivos del juego	1/05/20	1.2

Tabla D.2: Tabla de control de cambios del GDD

D.3. Descripción general

D.3.1. Contexto e historia

Noxious Liquid (D.1) es un videojuego de plataformas para la consola NES en el que el jugador tiene la capacidad de controlar a un personaje, Blob, que ha nacido de una explosión causada en una central nuclear. Dicha explosión ha esparcido elementos de la misma sustancia de la que estás compuesto: Plutonio.

El objetivo es recoger, a través de todos los niveles, todos los trozos de plutonio: Si no se recogen todos los trozos en un nivel se vuelve a empezar para que puedas recogerlos de nuevo. Si se recogen todos, se podrá pasar al siguiente nivel. No obstante, el camino hacia el final de los niveles no será fácil ya que, a parte de ti, existen los Evil Blob: Seres malignos que han sido creados en la misma explosión y que tienen especial interés por el plutonio.

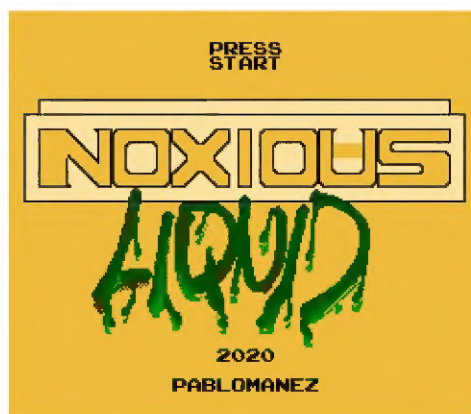


Figura D.1: Pantalla de inicio de Noxious Liquid

D.3.2. Funcionalidades generales

- **Sistema de colisiones:** Sistema para que el personaje pueda desplazarse libremente por la pantalla, con detección de colisiones, aplicación de gravedad y movimiento a través de la pulsación de los botones del mando.
- **Sistema de scroll en los niveles:** Sistema que permite, a partir de los datos almacenados en memoria, pintar nuevas columnas y que vaya progresando en cada nivel.
- **Sistema de cambio de niveles:** Junto a una progresión adecuada de éstos, es un sistema para elegir el orden de los niveles y todos los datos que necesitará cada uno de ellos.
- **Animaciones de los personajes:** A través de los sprites almacenados en la PPU se va a definir el comportamiento de una serie de animaciones predefinidas, según distintas situaciones en las que se encuentre el personaje.
- **Exportador personalizado de niveles:** Para facilitar el trabajo de creación de los niveles, se va a necesitar un programa que trate un archivo CSV con los datos de cada nivel y que en la salida proporcione todos los datos necesarios del nivel que se acaba de tratar.
- **Compresión de niveles:** Para poder crear muchos más niveles, hay que buscar una forma de comprimir los datos de cada uno de ellos.

D.3.3. Características de los personajes

- **Blob:** Compuesto por plutonio es el personaje principal que puede controlar el usuario. Tiene la capacidad de moverse y saltar. Si cae al vacío o toca algún enemigo, muere al instante. (D.2)
- **Evil Blob:** Los enemigos, rondando por todo el nivel, protegen los trozos de plutonio. (D.3)



Figura D.2: Personaje principal

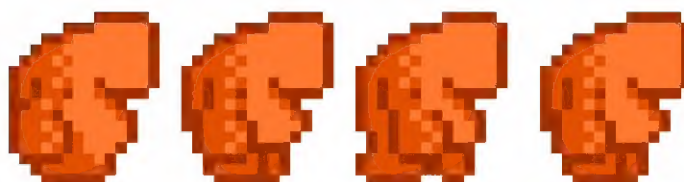


Figura D.3: Enemigos

D.3.4. Escenarios

- **Central nuclear:** Es el escenario principal compuesto por zonas desoladas por la explosión y tuberías destrozadas con apenas lugares para avanzar.

D.3.5. Restricciones

Teniendo en cuenta que este videojuego es mi trabajo de fin de grado, en cuanto a personal para desarrollar el juego solo estoy yo y en los plazos para su realización son limitados: Comenzado en octubre de 2019 hasta mediados de mayo de 2020.

D.4. Requisitos específicos

D.4.1. Mecánicas de los jugadores

Tabla D.3: Mecánicas de los jugadores

Identificador	Título	Descripción
MJ1	Caminar/Correr	Movimiento a izquierda o derecha del personaje siempre y cuando pueda realizarse, colisione o no con algún bloque.
MJ2	Saltar	Impulso variable que aumenta la altura y permite esquivar o llegar a nuevas alturas del nivel.
MJ3	Morir	El personaje morirá en el momento que caiga al vacío o choque con algún enemigo que se interponga en su camino.

D.4.2. Mecánicas de los de los objetos y NPCs

Tabla D.4: Mecánicas de los objetos y NPCs

Identificador	Título	Descripción
MN1	Caminar/Correr	Movimiento a izquierda o derecha del personaje. En cuanto se detecta colisión con algún bloque, éste cambia la trayectoria hacia el lado contrario

D.4.3. Técnicas y algoritmos a desarrollar

Tabla D.5: *Técnicas y algoritmos a desarrollar (continuación)*

Identificador	Título	Descripción
T1	Sistema de colisiones	Sistema que permite a cualquier entidad del juego la capacidad de no poder moverse a cierta posición debido a la colisión con un "bloque" predefinido que contiene ésto. Su implementación viene dada gracias a que existe un mapa de colisiones, predefinido para cada nivel, a nivel de bit para evitar malgastar memoria
T2	Sistema de scroll	A través de la llegada del personaje a cierta región de la pantalla, se activa este sistema que modifica ciertas variables y permite el dibujado de nuevas columnas de sprites dando la sensación de que se está recorriendo un nivel totalmente completo
T3	Exportador personalizado	Programa personalizado, escrito en C++, al que se le pasa como entrada un archivo .csv y escribe tres archivos distintos: Uno para el mapa, con el índice de cada sprite que hay que usar, otro para los atributos, con las paletas que deben usarse en cada momento y en cada zona de la pantalla, y uno último para las colisiones. Los datos relacionados con cada índice, colores y colisión están predefinidos en el propio programa. Cabe decir, que el achivo de entrada es conseguido mediante la pre construcción del nivel en Tiled con los mismos sprites que se usarán en el definitivo.
T4	Compresión de niveles	Sin compresión, cada nivel ocupa un total de 8kb de memoria, o un banco de memoria, por lo que se ha comprimido el almacenamiento a través del uso de metasprites: Cada uno de ellos almacena la información de cuatro sprites individuales.

D.4.4. Requerimientos no funcionales

Tabla D.6: Requerimientos no funcionales

Identificador	Título	Descripción
---------------	--------	-------------

Tabla D.6: *Requerimientos no funcionales (continuación)*

Identificador	Título	Descripción
NF1	Estética retro y agradable de ver	Se pretende aportar una estética retro al videojuego y el uso correcto de la variedad de colores que aporta la NES.
NF2	Compatibilidad para sistemas NTSC y PAL	Puesto que ambos sistemas trabajan a distinta tasa de refresco, se trabaja en optimizar todas las rutinas del juego.

D.4.5. Restricciones

Aunque la NES permita el uso de más memoria de lo normal, gracias al cambio de bancos y distintos mappers, la memoria que ésta aporta no es algo muy grande, por lo que hay que llevar cuidado con la información que se duplica y todo aquello que se almacena en la ROM. El rendimiento tampoco es algo que prime en una consola de 30 años, por lo que cada frame está limitado a ciertos ciclos. Además, la programación del videojuego se está llevando a cabo en ensamblador 6502 y no en un lenguaje de programación moderno.